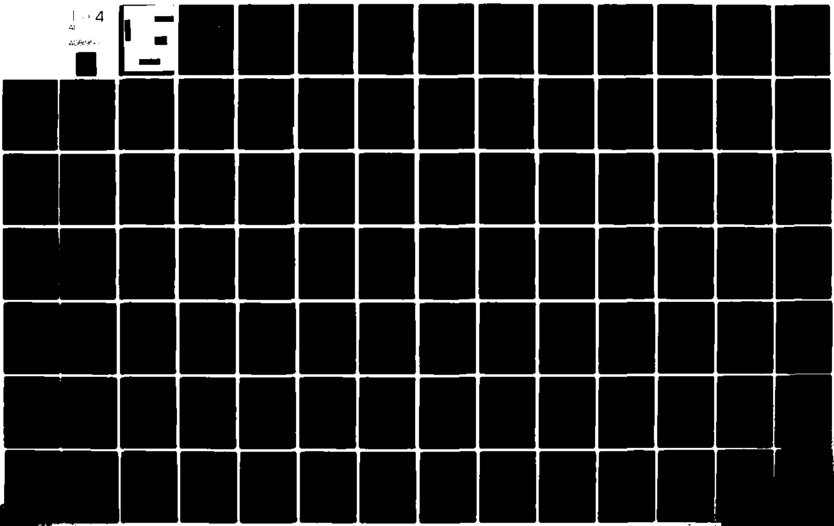
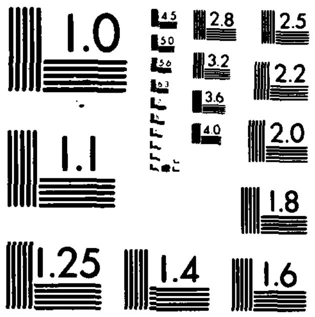
AD-A081 950 MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 9/2
THE EVENT BASED LANGUAGE AND ITS MULTIPLE PROCESSOR IMPLEMENTAT--ETC(U)
JAN 80 A REUVENI N00014-75-C-0661
UNCLASSIFIED MIT/LCS/TR-226 NL

1 of 4
AD A081950





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TR-226

THE EVENT BASED LANGUAGE AND ITS MULTIPLE PROCESSOR IMPLEMENTATIONS

Asher Reuveni

This research was supported by
The Advanced Research Projects Agency
of the Department of Defense
and was monitored by the Office of Naval Research
under Contract No. N00014-75-C-0661

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER MIT/LCS/TR-226	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 The Event Based Language and Its Multiple Processor Implementations		5. TYPE OF REPORT & PERIOD COVERED Ph.D. Thesis - January 1980
		6. ~~PERFORMING ORG. REPORT NUMBER~~ 14 MIT/LCS/TR-226 ✓
7. AUTHOR(s) 10 Asher/Reuveni		8. ~~CONTRACT OR GRANT NUMBER(s)~~ 15 N00014-75-C-0661
9. PERFORMING ORGANIZATION NAME AND ADDRESS MIT/Laboratory for Computer Science ✓ 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS ARPA/Department of Defense 1400 Wilson Boulevard Arlington, VA 22209		12. ~~REPORT DATE~~ 11 Jan[redacted]80
		13. ~~NUMBER OF PAGES~~ 290
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) ONR/Department of the Navy Information Systems Program Arlington, VA 22217 12 290		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document has been approved for public release and sale; its distribution is unlimited

[illegible]

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

9 Doctoral thesis

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

programming languages	event handlers	NP-complete
parallel programming	EBL	
concurrency	modularity	
synchronization primitives	networks	
events	data flow	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This research defines and analyzes a simple language for parallel programming which is designed for multiple processor systems. The language (EBL) is based on events which provide the only control mechanism. Events are explicitly caused by the program, and they activate instances of dynamic program units called event handlers. The only operation that can be performed by an instance of an event handler is the causing of new events. The language constructs are primitive; nevertheless, the capability of hierarchical program design is provided via static modules and other modularity sources.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

409648 [illegible]

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20.

The language does not contain conventional constructs such as: variables, assignment statements, goto statements, iteration constructs, procedures, functions, and semaphores; however, these can be easily modeled. In addition, events allow activation of parallel processes, synchronization of parallel processes, mutual exclusion, message passing, immutable objects, and the effect of mutable objects.

Schemes for implementation of the language on processor networks are investigated. An implementation scheme based on communicating managers which operate without any centralized control is described. A relaxed distributed locking algorithm in which deadlocks are prevented is developed; it does not assume a total order on all objects to be locked. Several optimization problems, e.g., optimal distribution of objects in a network, are investigated. The problems are shown to be *NP*-hard and heuristic algorithms are suggested. Implementation schemes of the language on a data flow processor are described. These add to the processor the capability of procedures, and synchronization primitives such as semaphores.

THE EVENT BASED LANGUAGE AND ITS MULTIPLE PROCESSOR IMPLEMENTATIONS

Asher Reuveni

January 4, 1980

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0661.

Massachusetts Institute of Technology
Laboratory for Computer Science

Cambridge Massachusetts 02139

THE EVENT BASED LANGUAGE AND ITS MULTIPLE PROCESSOR IMPLEMENTATIONS

by

ASHER REUVENI

Submitted to the Department of Electrical Engineering and Computer Science on January 4, 1980 in partial fulfillment of the requirements for the Degree of Doctor of Philosophy.

Abstract

This research defines and analyzes a simple language for parallel programming which is designed for multiple processor systems. The language (EBL) is based on events which provide the only control mechanism. Events are explicitly caused by the program, and they activate instances of dynamic program units called event handlers. The only operation that can be performed by an instance of an event handler is the causing of new events. The language constructs are primitive; nevertheless, the capability of hierarchical program design is provided via static modules and other modularity sources.

The language does not contain conventional constructs such as: variables, assignment statements, goto statements, iteration constructs, procedures, functions, and semaphores; however, these can be easily modeled. In addition, events allow activation of parallel processes, synchronization of parallel processes, mutual exclusion, message passing, immutable objects, and the effect of mutable objects.

Schemes for implementation of the language on processor networks are investigated. An implementation scheme based on communicating managers which operate without any centralized control is described. A relaxed distributed locking algorithm in which deadlocks are prevented is developed; it does not assume a total order on all objects to be locked. Several optimization problems, e.g., optimal distribution of objects in a network, are investigated. The problems are shown to be *NP*-hard and heuristic algorithms are suggested. Implementation schemes of the language on a data flow processor are described. These add to the processor the capability of procedures, and synchronization primitives such as semaphores.

Name and Title of Thesis Supervisor:

Stephen A. Ward,
Associate Professor of Electrical Engineering and Computer Science

Key Words and Phrases:

programming languages, parallel programming, concurrency, synchronization primitives, events, event handlers, EBL, modularity, networks, data flow, *NP*-complete

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisor, Professor Steve Ward, for his support and encouragement. Steve's desire for simplicity, and constructive criticism were most valuable. My thesis readers, Professors Carl Hewitt and David Reed, provided many helpful suggestions that improved the content and representation of the thesis. Professor Adi Shamir is gratefully acknowledged. The relatively few discussions we had were most fruitful and stimulating.

Many fruitful discussions with Chris Cesar and Al Mok of the Real Time Systems group at M.I.T. helped in crystallizing some of my ideas. The members of the Real Time Systems group at M.I.T. which maintained the UNIX system used to produce this document are acknowledged. Clark Baker, who developed the software used to automatically include drawings in documents, and Chris Cesar helped by solving many problems encountered during the editing of this document.

My wife, Sara, deserves my warmest thanks and gratitude for her love, patience, and encouragement during the last three years. By virtually assuming the responsibility for taking care of our two children and all daily problems, she created a most convenient and pleasant environment. Her task was not easy especially due to the long distance from M.I.T. to our home country. Our parents deserve special thanks for encouraging me to pursue my doctoral studies despite the knowledge that as a side effect they will miss us for a few years.

CONTENTS

FIGURES

1. Introduction

The research described in this thesis deals in the first part with EBL, an event based language for parallel programming which is designed for multiple processor applications, and in the second part with strategies for its implementation on multiple processor systems.

1.1 Motivation

The decreasing cost of computer hardware will cause building of more and more multiple processor systems of various kinds. Examples of such systems are: C.mmp [Wu-72], Pluribus [He-73], Data Flow [De-75], the boolean n-cube parallel machine [Su-77], and the MuNet [Wa-78b]. However, most of today's programming languages are not designed to exploit the concurrent processing capabilities offered by such systems.

The classical languages such as ALGOL, or FORTRAN do not support parallelism at all. PL/1 has a multi-tasking capability, therefore one can easily express in the language a process spawning concurrent processes. Unfortunately, the language lacks adequate primitives for synchronization of concurrent processes [Mo-76]. It does not provide, for example, any mechanism supporting mutual exclusion; thus, to achieve mutual exclusion among concurrent processes one must resort to some form of busy waiting, a solution which wastes computational resources.

More modern languages such as MODULA [Wi-77b], CONCURRENT PASCAL [BH-75], and communicating sequential processes [Ho-78a] provide for parallelism (as well as synchronization primitives), but only in the limited form of sequential processes. These languages have certain disadvantages: In MODULA only the main process can activate other processes, thus the rate of spawning new activities is limited. Communicating sequential processes is a static language. The maximum number of concurrent processes is bounded, and the set of processes with which a process can communicate is fixed and can be found from the program text.

Languages based on message passing models of computation such as the Actor model [He-76] do not suffer from the above disadvantages. Our model is reminiscent of message passing models but there are some fundamental differences which are discussed in section 1.5.

1.2 Research Goals

The language design goals have been:

1. High expressive power and universality, especially suitable for expressing parallel computations in a distributed processing environment.
2. A small number of constructs, even to the extent of obtaining only a base language.
3. The capability of hierarchical (top down) program design, and the ability of creating encapsulated program units that can be designed and checked in a modular way. Such features are especially important in a language for parallel programming.
4. A reasonable implementation; although the language has not been implemented, implementation issues influenced many of the decisions made during the design.

Another goal of this research has been the investigation of strategies for implementation of the language on multiple processor systems. Our primary interest has been the exploitation of concurrency within programs; therefore, we have concentrated on two types of systems which seem suitable: a processor network, and a data flow processor [De-77]. A processor network is attractive not only because of its potential computing power, but also because of its scaling characteristics. In contrast to conventional systems, a processor network is not limited by dependencies on shared resources which become bottlenecks as the system's size increases [Wa-78b]. The data flow processor has been selected since it is designed to achieve a highly parallel operation. Our implementation schemes are not restricted to the above types of systems; they can be easily adapted to other systems.

1.3 Our Approach

Our approach to the language design was partially motivated by our desire to investigate the intrinsic power of events. We asked ourselves whether a language in which events provide the only control mechanism can be designed, or whether additional constructs are essential. In order to find the answer we tried to see what is the inherent power of events: what do they capture, what computations can be expressed in a language using only events, and which conventional control structures can be expressed in terms of events. We approached the problem by going almost to the extreme case. The defined language is very condensed and it has no conventional control structures or data structures. In particular, the language does not contain variables, assignment statements, goto statements, iteration constructs, procedures, or functions.

Our approach to implementation of the language has been to exploit the various sources of parallelism in a program. For this reason schemes relying on centralized control have not been considered. Instead, our basic implementation scheme involves many managers communicating with each other. The role of a manager is limited; each manager handles one event class or one (normally small) program unit.

Distributed implementations are harder than single processor implementations. One encounters more difficulties when implementing a distributed database than in the case of a conventional (single processor) database. The price paid for achieving higher concurrency (as well as other advantages) in the case of a distributed database is that some of the operations require special mechanisms for their correct implementation. For example, locking of objects, and maintaining consist·.ncy are more difficult to implement in a distributed system. It is not surprising, therefore, that similar problems have been encountered in the course of designing implementation schemes for our language. Various techniques have been devised in this research for solving these problems.

1.4 The Event Model

Other formalizations of the event notion have been done by several message passing models of computation, e.g., the Actor model [He-76], and the mu calculus [Ha-78, Wa-78a]. Their notion of event differs from ours and the differences are discussed in section 1.5. The following description of our model is not formal; our language formalizes it. Only the fundamental properties of our model are described in this section; many of the features are abstracted (some of them are given in the next chapter) and only those needed for comparison with other models of computation are given.

In our model, events are abstract entities that are explicitly caused by the program during the course of the computation. An event object is created upon each occurrence of an event and it carries information on the nature of its occurrence. The occurrence of an event can be remembered forever, i.e., the corresponding event object is permanent, or can be forgotten at some stage of the computation, i.e., the corresponding event object is *temporary*. The distinction between these two kinds of behavior is done syntactically, on the basis of explicitly declared types. In the former case, an event may have several direct effects and it is called a multi_use event. In the latter case, it may have at most one direct effect after which it is forgotten; in this case the event is a single_use event. A direct effect of an event is the activation of an instance of a program unit (an *event handler*). In our model the program consists of fixed program units (the *number of program units* and their identities are fixed); an instance of a program unit is activated when several events satisfying some condition specified by the program unit have occurred. Once an instance of a program unit is activated it causes some events and then vanishes.

The occurrence of an event is brought to knowledge of (broadcast to) all program units and thus may activate instances of more than one program unit (if the corresponding event object is permanent), each causing more events. The information about the nature of an event occurrence cannot be modified: it either remains unchanged forever (if the event object is permanent), or is totally *forgotten* (if the event object is temporary). The latter happens when the event activates an instance of a program unit; one of the effects of such activation can therefore be thought of as *uncausing* (forgetting) the event. The ability to uncause an event introduces mutable objects to the model, and this will be examined in section 3.2.

An instance of a program unit causing an event does not know and has no control over: what program units will be affected by it, how many instances of program units will be affected, and when will they be affected. Each program unit knows which events have occurred so far and thus can autonomously decide which of them is relevant to it and should affect it. In this respect, program units are reminiscent of daemons (event driven procedures; see for example [Pf-74, Wi-77a]), or knowledge sources of Hearsay [Le-75].

In our model as presented so far, there is basically one global environment for events accessible to all program units. This is different from models supporting local (or nested) environments such as block structured languages, the lambda calculus, or the mu calculus, where the structure of the program determines the various environments.

The main advantage of a global environment is that every program unit can access all objects, and thus all program units can communicate with each other. Such unrestricted access capabilities add to the expressive power of the language, e.g., in comparison to the mu calculus where a message can be only sent to nested receivers. On the other hand, the lack of local environments has several disadvantages such as: lack of modularity on its various aspects (e.g., local name spaces, local objects, protection of objects, abstract data types, or encapsulated program units), or lack of temporary objects (i.e., bad storage utilization).

Our language manifests some compromise between one global environment and local environments. It contains several mechanisms (e.g., modules, event classes, formal parameters) which eliminate some of the disadvantages of a global environment while allowing the programmer to control the extent to which objects are accessible to program units.

Modularity is manifested in several forms in the language. The first source of modularity stems from our modules (which are defined in the next chapter). Modules allow hiding and sharing of identifiers between program units, protection of object classes (event classes) against undesired uses, creating encapsulated program units and abstract data types, top down and bottom up designs.

Another source of modularity (better called freedom) stems from the fact that the order of program units (event handlers) in the program has no effect on the meaning of the program. This relaxes the constraint of most existing programming languages where consecutive steps in an algorithm are typically adjacent in the program text (except where transfer of control occurs). In our case, the algorithm for handling an event can be distributed in the program text. This freedom does not necessarily contribute to program modularity; however, it allows the program to be organizeded in some meaningful structure. For example, a real time program for controlling a physical system, such as a car, can be constructed as a collection of modules that reflects the structure of the physical system; each program module will correspond to some physical module. On replacement of one physical module, only the corresponding program module has to be modified.

1.5 Relation to Message Passing Models

A possible way to understand this model is by describing its behavior in terms of message passing models such as the Actor model [He-76], or the mu calculus model [Ha-78, Wa-78a]. The basic computational step in a message passing model is passing a message from one instance of a program unit to another instance of a program unit. In the event model, the basic computational step is causing an event by one instance of a program unit. This can be thought of as broadcasting a message to some *ether* enclosing all program units (akin to ETHER [Ko-79]). Another way to understand it is as writing something on a *blackboard* accessible to all program units (akin to Hearsay [Le-75]).

Our basic computational step does not include the receipt of the message by other program units since no explicit target is associated with the message. Therefore, our basic computational step consists only of broadcasting the message to the ether. The message describes the nature of the event occurrence; it carries both values and control information. A program unit (conceptually) continuously examines the ether and when it finds a collection of relevant messages, an instance of the program unit is activated.

We distinguish between permanent and temporary messages. A message of the first kind cannot be removed from the ether (erased from the blackboard); it can be examined by more than one program unit, and cause activation of several instances of program units. A message of the second kind is deleted from the ether when it causes the activation of an instance of a program unit. In both cases, a message in the ether is an *immutable object* since it describes the nature of a past event.

A program unit can remove from the ether one or more messages (uncause one or more single_use events) in an atomic action. This allows easy solutions to synchronization problems such as the five dining philosophers (see chapter 6). A program unit need not specify the order in which it processes messages (events) from a certain class; this can be left for the system. However, a program unit can specify some desired order, e.g., a FIFO order. Other orders can be specified and this allows easy solutions to scheduling problems such as the (minimum distance) disk head scheduler (see chapter 6).

Some of the fundamental differences between EBL and languages based on the Actor model, PLASMA [He-76] and Act1 [He-79], are discussed now.

1. The concept of event is different. In the Actor model, an event is defined as the *receipt* of a message by an actor, whereas in EBL's model, the occurrence of an event can be associated with the *transmission* of a message (an event object).
2. A message in the Actor model specifies an explicit target, whereas no target is specified by an EBL event. This allows the same message (event) to be detected by more than one program unit and thus adds to the expressive power of EBL. The price for this is paid by the receivers which have to find the messages in which they are interested.
3. Events in EBL (more precisely event classes) are named, while the Actor model has no mechanism for naming events or messages. This allows a program unit to specify the message classes in which it is interested. This capability adds to the expressive power but the tradeoffs are similar to those related to the lack of explicit targets.
4. EBL is strongly typed while PLASMA is not a typed language. There is a greater

programming freedom in a non-typed language; however, fewer checks can be done at compile time, and certain run time errors that could have been detected at compile time may occur in a non-typed language. Note that Act1 is a typed language.

5. Messages (event objects) in EBL are restricted, and must be of fixed types. In the Actor model, the structure of a message M sent from actor A to actor B is not restricted. The message M can be any actor; it can carry data or even an explicit algorithm that B can activate. A possible analogue in EBL would be allowing an event handler as a parameter of an event. We have not included such a capability in EBL because it complicates the language and its usefulness is not apparent.

6. In EBL there is a complete separation between algorithms and state information (control or data) they need for their execution. Algorithms are represented by event handlers which are therefore *pure*, and all state information is represented by events. In the Actor model the separation exists only for unserialized actors since a serialized actor has a state associated with it. In general, a pure algorithm has an advantage in a distributed system since multiple copies of it can be kept in the system and each can be activated whenever the information it needs arrives.

7. EBL's feature of one (multi_use) event directly activating several instances of event handlers, has no analogue in PLASMA and Act1. ETHER [Ko-79], which is also based on the Actor model, has a similar capability.

8. PLASMA and Act1 are based on a message passing model of computation, but they both rely on other control mechanisms (e.g., a case statement) in addition to

message passing. EBL is based on events and uses them as the only control mechanism

One advantage of a message passing model over a procedure based model is achieved by using *continuations* [St-74, He-76, Ha-78, Wa-78a]. Instead of waiting for the result of a procedure, the program unit replacing the procedure is supplied with an argument specifying where to send the result. The activator can then relinquish its computational resources and vanish instead of waiting for the result holding computational resources [He-76, Wa-78a]. A similar scheme can be used in our event model which therefore shares the same advantage over the procedure based model. The price paid for this early release of computational resources (in a message passing model or in the event model) is that the state of the computation must be passed to the continuation, e.g., by including it as additional arguments to the program unit replacing the procedure (which then passes it to the continuation).

1.6 Our Main Contributions

The fundamental characteristic or our event model is the ability of an instance of a program unit (an instance of an event handler) to unilaterally broadcast messages (cause events) without specifying their targets. The receivers (event handlers) autonomously decide whether they are interested in the messages or not. The receivers can remove from the ether, which encloses all program units, one or more messages in an atomic action.

In our event model, the primitive objects (event objects or messages) are not mutable objects since once a message is broadcast to the ether it cannot be modified. If it is a temporary message, it can only be deleted and totally forgotten; thus, it is not viewed as a mutable object. There are, however, unique ways to model mutable objects in EBL; the effect of mutable objects can be achieved due to:

1. The ability to broadcast permanent messages belonging to specific classes (cause multi_use events), and to read the latest message from a class.
2. The ability of program units to remove messages from the ether (uncause or forget single_use events) and to broadcast new messages from the same classes.

The language does not contain conventional constructs such as: variables, assignment statements, iteration constructs, procedures, functions, and semaphores; however, all these constructs can be easily modeled. The high expressive power of the language is mainly due to our single_use events which are the work horse of the language.

The implementation schemes developed in this thesis are not conventional implementations of programming languages. The basic implementation scheme associates an event class manager with each event class, and an event handler manager with each event handler in the program. The managers communicate with each other and operate without any centralized control.

Some of the problems encountered during the design of the managers algorithms are reminiscent of problems encountered in the implementation of a distributed database; in particular, a distributed locking algorithm. Our locking algorithm is a two phase algorithm in which deadlocks are prevented. In contrast to many existing algorithms which prevent deadlocks by defining a total order on all objects to be locked, our scheme only defines a partial order on all object classes. The advantage of this scheme is that objects can be locked by a requestor concurrently and not sequentially as in other algorithms.

Several optimization problems which are of general interest, e.g., optimal distribution of objects in a network, have been defined and investigated. We have proved that the optimization problems and even approximations to these optimizations are *NP*-hard, and suggested heuristic algorithms.

Even though the data flow processor [De-77] supports highly parallel operation, it was not designed to support communicating sequential processes. Several modifications to the basic data flow processor have been suggested; these increase the efficiency of executing a program consisting of several processes on the data flow processor. An implementation of our language on the data flow processor adds to the processor the capability of procedures and synchronization primitives such as semaphores.

1.7 Plan of the Thesis

The first chapter of this thesis describes our event model and its relation to message passing models. Our approach to the language design and to the implementation has been described. Chapter 2 gives a quite detailed overview of the language. Chapter 3 is intended to give a deeper insight into the language. This is done by analyzing some of the main properties of the language more thoroughly, and by comparison to other languages and models. Chapter 4 completes the definition of the language. Chapter 5 deals with the expressive power of the language. It shows how conventional language constructs can be expressed in the language. Chapter 6 gives some classical examples which illustrate the ease of expressing various programs in the language. These examples lead to several interesting observations.

Chapters 7-9 are devoted to strategies for implementation of the language on multiple processor systems. Chapter 7 investigates implementation schemes which are natural to the language. A system with virtually unlimited computational resources is selected as a concrete example. This system allows us to abstract some of the limitations posed by more restricted computer systems and thus to concentrate on the fundamental problems. Chapter 8 investigates the problems imposed by more real processor networks. The effects of limiting the number of available processors, dealing with a processor network of a given configuration (not necessarily a complete graph), and limiting the number of neighbors of each processor, are studied. Chapter 9 examines the possibility of implementing the language on a data flow processor [De-77]. The data flow processor is designed to achieve a highly parallel operation and is therefore a natural candidate for implementing our language.

Chapter 10 summarizes the thesis, presents our conclusions, and makes suggestions for further research. Chapter 11 contains the references. Appendix A contains a formal definition of the syntax. Appendix B contains performance evaluation of a program on a processor network, and appendix C contains performance evaluation of a program on a data flow processor.

Let us give some suggestions related to reading of this dissertation. A reader who is only interested in the language can read chapters 2-6. Appendix A can be read concurrently with chapter 4; it can also be used as a reference. A reader who is primarily interested in implementation issues has first to read chapter 2, and section 4.6.1; these give sufficient background for understanding our implementation schemes. Chapter 7 should be read by anyone who is interested in implementation schemes of the language. It is prerequisite for understanding chapter 9. Chapter 9 can be skipped by a reader who is not familiar with the data flow processor. Chapter 8 can be read in two different ways: An implementation oriented reader can read it and skip the proofs; he can view the chapter as a continuation of chapter 7. A theorist, however, who is interested neither in the language nor in the implementation schemes, can read chapter 8 without first reading any other chapter of this thesis. Performance evaluation of EBL programs appears in appendixes B and C. Appendix B can be read after reading chapter 8 (without the proofs of chapter 8). Appendix C relies on appendix B and can be read after reading chapter 9.

2. Overview of EBL

This chapter contains a somewhat detailed overview of the language. The main features of the language are described here without the annoying little details.

2.1 Events and Event Classes

Event is the main data type of EBL. *Events* are abstract entities that are *caused* during the course of the computation. An event is *caused* either *internally* and *explicitly* by the program, or *externally*, when some hardware device causes it. (In order to avoid cumbersome descriptions, in many places throughout the thesis the word event stands for an occurred event.) An *event object* is created upon each occurrence of an event and carries some information about the nature of its occurrence. This information consists of explicit values, in the form of *event parameters*, and implicit control information. Every event is a member in some *event class*, denoted by an *event class identifier*; event class identifiers are declared. All events from an event class are of the same type, the type associated with the corresponding event class identifier. The number of the parameters and the types of corresponding parameters are identical for all events from the same class.

An event parameter can be of a simple type (e.g., **int**, **bool**, or **char**), or of *event type*. The value associated with such parameter is either a value of a simple type, or an event class identifier. For example, if E1 is an event class identifier, then E1(1) and E1(2) are examples of possible events from class E1. Events from another class, E2, can be E2(E1,3,**true**), and E2(E1,5,**false**). The parameters of the latter event are: the event class identifier E1, the integer value 5, and the boolean value **false**.

An event in EBL is either of a **multi_use** type (a **multi_use** event), or of a **single_use** type (a **single_use** event). In the former case, its occurrence is remembered forever, i.e., the corresponding event object is *permanent*. In the latter case, its occurrence can be forgotten at some stage of the computation, namely, the corresponding event object is *temporary*. In both cases, the event object created upon the occurrence of the event is an *immutable* object. The main difference between the two types of events is in the number of direct effects that they can have. A *direct effect* of an event is the activation of an instance of some program unit (to be defined shortly). A **multi_use** event can have several direct effects, while a **single_use** event can have at most one direct effect, after which it is forgotten. We say that an event *exists* when its object exists.

An orthogonal property of events is their ability to *recur*. Events can be either **recurrent** or **non_recurrent**. In the former case events can recur, in the sense that at any point in time the conceptual list of event objects from the corresponding event class, the *event list*, can contain identical objects. This list is therefore a *collection* in the sense of [He-76]; i.e., a multiset. In the latter case events cannot recur, in the sense that trying to cause an event while an identical event is remembered has no effect. In this case, at any point in time the conceptual list cannot contain identical objects, and therefore corresponds to a *set*. The term **recurrent** stems from the possibility of referring to several identical events as several instances of the same event; however, the term event instance will not be used here.

2.2 Event Handlers

A program unit in EBL is called *event handler*. An instance of an event handler is activated upon occurrences of events from certain event classes. The event handler consists of two parts: the *event handler heading*, and the *body*. The *event handler heading* contains several *event descriptors* each of which specifies an event class and contains formal parameters (if the type of the event class contains parameters), and a *where clause* which specifies a condition (a slightly extended conventional boolean expression) that has to be satisfied for each activation of an instance of the event handler. A collection of *existing* events that *match* the event handler heading, namely, the collection contains one event for each event descriptor in the heading and the condition is satisfied, may cause the activation of an instance of the event handler.

The *body* of the event handler consists of two parts: the *declaration part* and the *script*. The declaration part can only contain declarations of special identifiers of type tag (defined in section 2.3); event class identifiers can be declared only outside event handlers. The script which defines the behavior of the event handler has a very limited structure; it only contains a list of events to be caused. The following example of an event handler shows two event descriptors in the event handler heading: sum1(i: int, t1: tag), and sum2(j: int, t2: tag) corresponding to the event class identifiers sum1 and sum2 respectively; each contains two formal parameters. The heading also contains the condition: i>j. The script specifies two events to be caused: one from the event class next, and the other from the event class print.

```
on sum1 (i: int, t1: tag) ∧ sum2 (j: int, t2: tag)  where i>j
   seq_cause
          next (i+1) ;
          print (i+j)
end ;
```

An instance of an event handler is activated *at most once* for every combination of *existing* events that match the event handler heading. An event is considered "*used*" after it is selected for such activation. If it is of a **single_use** type it disappears after this use and therefore cannot be used for additional activations (this is one of the reasons why we said "at most once", and not just "once", at the beginning of this paragraph). If the event is of a **multi_use** type, it is not affected by its use. In order to avoid possible ambiguities, the activation of an instance of an event handler (in particular, the selection of the events, and the evaluation of the condition in the where clause) is defined as an atomic action.

The same event class identifier can appear in the headings of several event handlers. Thus, if it is of a **multi_use** type, an event from its class can activate instances of several event handlers. This feature is discussed further in chapter 3.

The execution of the script of an event handler involves causing the events specified in the script. These events may refer to parameters of the events that activated this instance of the event handler by means of formal parameters in the heading (see i and j in the example above). The actual parameters (which are expressions) of each event in the script are evaluated and the event is caused (an event object is added to the corresponding event list); this is the place where the actual computation takes place. These events can be caused either sequentially or concurrently (in parallel) depending whether the body starts with the keyword **seq_cause**, or **par_cause** respectively; the event handler is a *sequential handler* or a *parallel handler* respectively. For each activated instance of the event handler in the previous example, an event from class next is caused before an event from class print. If the event handler is transformed to a parallel handler then these two events can be caused in any order or even concurrently.

Let us digress a little to examine why a sequential handler is needed (a parallel handler simply allows expressing more parallelism). The ability to cause events sequentially is important. One may wish for example to cause several events after entering a critical region and then (sequentially) cause an event indicating termination of the execution of the critical region. In another example, one may wish to print the numbers 3 and 5 in this order. Assuming printing a number is triggered by causing an appropriate event, two events must be *sequentially caused* to achieve the desired effect.

Since the events caused by a parallel handler are not ordered, the only way (or slight modification of which) to sequentially cause events without using a sequential handler, is that each event in the sequence (except the last one) directly activates an instance of an event handler which (directly or indirectly) causes the next event in the sequence. Unfortunately, such a scheme does not work when the events in the sequence are **single_use** events. If they succeed to activate instances of the event handlers needed to *form* the sequence, all but the last in the sequence cannot have any other direct effect (by definition of a **single_use** event); in particular, the effects originally expected (e.g., printing 3). If any of them succeeds to have another direct effect, then the sequence is broken. Using a scheme in which the event handler whose instance should be activated as a direct effect of an event in the sequence also causes (directly or indirectly) the next event in the sequence is possible in some cases. However, such a scheme is too cumbersome and may result in inefficient programs in many cases. Thus, the need to cause a sequence of **single_use** events is the fundamental reason for the inclusion of a sequential handler in the language.

The condition in the where clause of the event handler heading contains a conventional boolean expression that has access to the formal parameters of the event descriptors in the heading. It is used to specify some conditions, or constraints on the activating events. Special *predicates* for specifying additional constraints on the order in which events are selected for activating instances of an event handler, can be used in the where clause of an event handler; they are defined in chapter 4.

2.3 Tag Identifiers

The language also contains the simple type **tag**. Identifiers of type **tag**, *tag identifiers*, can be declared outside event handlers, or within event handlers (as local identifiers). The values associated with identifiers of type **tag** are obtained from a set of abstract unique values: α, β, ... , called the *tag set*. This is a global set in the sense that there is only one such set for the whole program. For each activation of an instance of an event handler its local tag identifiers assume unique values from the tag set. Once a value from the tag set is associated with a tag identifier, it is deleted from the set and cannot be associated with any other tag identifier.

Tag identifiers can be used to distinguish between the effects of different instances of event handlers. This can be done by *tagging* an event, namely, attaching to it a parameter of type **tag**. This feature has an important use when trying to join several events belonging to the same logical computation and activate an event handler. It can be done simply by specifying in the where clause of the corresponding event handler that they all have the same tag (see example in Figure 2.1).

2.4 Type Identifiers

The language is strongly typed; each identifier in the program must be declared as being of some type. In order to make life easier for the programmer, the definition of type identifiers is allowed. A *type identifier* can serve as a synonym or a shorthand notation for an explicitly written type, or type list. For example,

```
cont == single_use recurrent event (int, tag) ;
```

defines cont as a type identifier (not as an event class identifier), and

```
record == int, int, bool ;
```

defines record as an identifier equivalent to a list of three simple types.

The ability to define type identifiers might seem just a syntactic sugar; however, as shown in the next section, this concept is needed for defining abstract data types and therefore has an important semantic role.

2.5 Modules

After the properties of events and the program units they activate have been described, the way a program is structured will be explained. First, event handlers cannot contain local event class identifiers or other event handlers. However, the *module* is a mechanism in the language allowing the use of local identifiers.

The *module* is similar to that of MODULA [Wi-77b]. This construct is a block having two special lists at its beginning (the *module interface*): the *import list*, and the *export list*. The *import list* is a list of identifiers declared outside the module that should be known inside the module. Similarly, the *export list* is a list of identifiers declared inside the module that should be accessible outside the module. Thus, the module allows the user to have some control on the scope of identifiers. A module can contain other modules or event handlers; however, event handlers cannot contain other event handlers or modules (their only local identifiers are the formal parameters and the tag identifiers).

The module provides some protection for event class identifiers. If an event class identifier is declared in some module, the only event handler headings in which it can explicitly appear are of handlers contained in this module or in inner modules that import the event class identifier. Only such handlers can use events from this class. However, *all* event handlers whose script is in the scope of that event class identifier, in particular, handlers out of the above module that know about the event class identifier via the exporting - importing mechanism, can explicitly cause events from its class.

An important property of the module is that when a type identifier is exported, only its name is known outside the module. The structure of the denoted type is not known outside the module and therefore event handlers outside the module cannot refer to components of an object from that type. If such a type is used as the type of a parameter of an event in some event class then only handlers within the exporting module can actually break such a parameter into its constituent basic type components and do some computations on them. Formal parameters outside the module can be bound to objects of such exported type; they can be inserted as parameters in newly caused events (i.e., be copied or passed on); such *formal parameters* can be compared to one another if they are of the same exported type; but they cannot be operated on outside the module in other ways. Thus, the module allows creating abstract data types, and encapsulated program units.

The module serves as an interface between the surrounding environment and its inner environment. However, in contrast to the event handler, the module has no dynamic effect; its only effects are the establishing of rules for scope and use of identifiers. The modules in a program are visible to the compiler, but they have no effect at run time. Their role is different from the blocks of ALGOL which have both static (local identifiers) and dynamic roles.

2.6 Programs

A *program* is a collection of declarations (of event class identifiers and tag identifiers), of type definitions, of event handlers, and of modules (that can contain declarations, type definitions, event handlers, and modules, etc.). The execution of a program starts by the system that causes an event from the class **program_start**. This can activate one or more instances of event handlers, that can cause other events, etc. All those activities take place concurrently. At any point in time there are some active instances of event handlers causing events, and there are conceptual lists of all existing events from each event class. The program is terminated at a point in time at which the computation cannot proceed and will not be able to proceed later; i.e., the following conditions are satisfied:

1. There are no active instances of any event handler.

2. No instance of an event handler can be activated (no matching event collections).

3. No spontaneously caused future system events (defined in section 2.8) will be able to activate any instance of an event handler.

4. No event of a class denoted by an event class identifier explicitly declared in the program (as opposed to a system event) can (or will) be caused by the system. Normally, if an event class identifier E is used as a parameter of a system event, events from class E can be caused by the system (as opposed to causing by an instance of an event handler).

2.7 Event Relations

Various partial orderings on events and event relations have been defined in [Gr-75] and [He-77], and many of the observations in this section rely on them. Each event handler H defines a strict *partial temporal ordering* on the events that activate an instance of H and those caused by that instance of H. The following two relations between events: the *causality relation* --c-> , and the *precedes relation* --p-> , give a better description of the order in which the different activities in a program take place. For a parallel handler:

```
on  $P_1(...) \wedge ... \wedge P_n(...)$  where  B
   par_cause
         $S_1(...)$ ;
         ...
         $S_m(...)$
end ;
```

let $p_1, \dots, p_n$ be a collection of events that activates an instance of the above event handler, and let $s_1, \dots, s_m$ be the events specified in the script for this instance of the event handler; then for each s_{j_k} caused by *this* instance of the event handler (remember that while a non_recurrent event exists an identical event cannot be caused), the following relations hold:

p_i --c-> s_{j_k} for all $1 \leq i \leq n$.

For a sequential handler:

```
on  P1(...) ∧ ... ∧ Pn(...)  where B
   seq_cause
          S1(...) ;
          ...
          Sm(...)
end ;
```

let $p_1, \ldots, p_n$ be a collection of events that activates an instance of the above event handler, and let $s_1, \ldots, s_m$ be the events specified in the script for this instance of the event handler. Let s_{j_1} be the first event in the latter list caused by *this* instance of the event handler, s_{j_2} the second, etc. and the last one s_{j_q} where $j_q \leq m$; then the following relations hold:

p_i --c-> s_{j_k} for all $1 \leq i \leq n$, and for all $1 \leq k \leq q$,

s_{j_k} --p-> $s_{j_{k+1}}$ for all $1 \leq k \leq q-1$.

Another rule which holds for any pair of events p and s is:

p --c-> s $\Rightarrow$ p --p-> s

This rule reflects the physical fact that if one event causes another one, then the second event must have been preceded by the first event. The above rules together with the following properties of the relations, completely define the relations: both --c-> and --p-> are nonreflexive transitive relations.

By applying the above rules, one can determine the strict partial temporal ordering between the events that activate an instance of an event handler and those caused by it; this ordering is simply equal to that obtained by the precedes relation. In general however, a strict partial ordering among all events caused by a program cannot be determined even if they are known. The reason is the nondeterministic nature of our model: the semantics of the language does not define the order of (or the algorithm for) choosing the next event handler to be activated. In cases of event class identifiers of a **single_use** type appearing in more than one event handler, an arbitrary choice is made and this implies that several strict partial orderings on the events caused by the program are possible. For example, consider the following program:

```
E1, E2, E3: single_use recurrent event ;

on  program_start
  seq_cause  E1
end ;

on  E1
  seq_cause  E2 ; E3
end ;

on  E1
  seq_cause  E3 ; E2
end ;
```

In this program, one event is caused from each of the classes E1, E2, and E3; let e1, e2, and e3 be these events respectively. One cannot determine whether e2 precedes e3 or e3 precedes e2 even though the fact that both occur is known. Simply saying that e2 and e3 are not ordered is not correct since this allows the possibility that they are caused in parallel. However, events caused by an instance of a sequential handler cannot be caused in parallel; they must be ordered by the precedes relation. Thus, two strict partial orderings are possible in this example.

Frequently, in specifying the behavior of a program or a subprogram, the *occurrence order* of some collection of events (e.g., all events in a certain event class) is referred to. This order reflects the total temporal ordering on the events under discussion. It is consistent with the transitive closure of the precedes relation defined above.

2.8 System Events

So far no way of interacting with the external world has been described. In fact, the language does not contain specific constructs for this aim. In each implementation of the language, some *system event class identifiers* will be defined for interacting with the standard I/O devices, with the special devices to be controlled, and in general with the operating system. The following is an example of a system event class identifier:

print: **single_use recurrent event (int)**

The semantics of such system event class identifier can be that the system prints the parameters of all events from the class print, according to their occurrence order.

2.9 Example

Figure 2.1 contains an example that illustrates some of the features of the language. The purpose of the example is not to demonstrate how programming in EBL is convenient, but rather, to incorporate many of the language features in one example. Examples demonstrating the ease of expressing various programs are given in chapter 6. The example describes a subprogram for computing concurrently factorial(i) and factorial(j). When both results are ready their sum is printed. This subprogram can be activated concurrently any number of times. The subprogram consists of two modules: one for computing factorial in an iterative manner, and another that uses the first one and prints the desired result; it is activated by causing an event of the form: sum_f(i, j). We assume that the system event class identifier print, described earlier, exists in the implementation of the language. Note that the braces {...} are used to indicate the beginning and the end of a comment.

```
          cont == single_use recurrent event (int, tag) ;       { type definition }

  module { factorial sum }
          export: sum_f ;
          import: cont, print ;
          sum_f: single_use recurrent event (int, int) ;
          sum1, sum2: cont ;

          module    { factorial }
                    export: F ;             { F can only be caused outside the module }
                    import: cont ;          { the structure of cont is known here }
                    F: single_use recurrent event (int, int, cont, tag) ; { declaration }

                    on F (n, p: int, c: cont, t: tag)  where n<=1
                       par_cause        c(p, t)     { cause the continuation event }
                    end ;

                    on F (n, p: int, c: cont, t: tag)  where n>1
                       par_cause        F (n-1, n*p, c, t)      { iterate }
                    end ;
          end ;     { of factorial module }

          on sum_f (i, j: int)     { when factorial sum is requested }
             t: tag ;
             par_cause          { activate the factorial module twice with same tag }
                    F (i, 1, sum1, t) ; F (j, 1, sum2, t)       { concurrently }
          end ;

          { when two matching results of factorial arrive print sum }
          on sum1 (i: int, t1: tag) ∧ sum2 (j: int, t2: tag)  where t1=t2
             par_cause          print (i+j)
          end ;
  end ;   { of factorial sum module }
```

Figure 2.1 Factorial sum

Several observations can be made about the example in Figure 2.1:

1. The loop is implemented by causing F within one of the event handlers activated by F. This is reminiscent of a recursive procedure call; nevertheless, no state of

the computation needs to be saved as a result of this inner cause; the analogy to tail recursion is appropriate.

2. Tags are used to join results of matching subcomputations.

3. Note the modularity of the program. F can be caused outside the inner module in which it has been declared but it cannot be used (in the sense of section 2.2) there. F is unknown outside the outer module since it does not appear in the export list of that module.

4. The factorial module is used in a modular manner. In order to invoke the computation of factorial(n) one has simply to cause an event of the form F(n,1,C,t) and to wait for the result event in another event handler. The result event has the form C(r,t) where r=factorial(n) if $n \geq 0$, else 1. One does not have *to know how the factorial module is implemented.*

5. sum1 and sum2 are used as continuations.

6. The type identifier cont is known in both modules since it has been imported appropriately. Its structural details are also known in both modules since they are inner modules in the environment in which cont has been defined.

Suppose the two events, sum_f(1,3) and sum_f(2,1), are caused sequentially. The corresponding strict partial temporal ordering between the caused events is depicted in Figure 2.2 (similar drawings appear in [He-77]). From the graph in Figure 2.2 it can be seen how four activities take place concurrently, and how the results of related activities are joined. Unordered events (e.g., print(3) and print(7)) can occur in any order or concurrently.

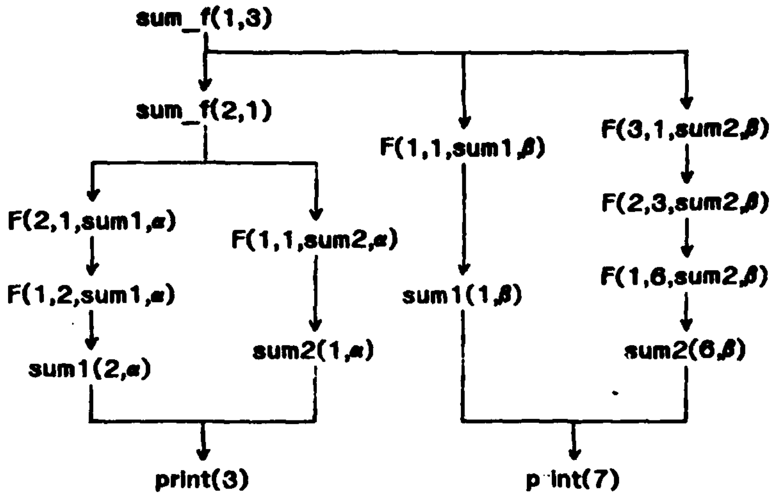


Figure 2.2 Events ordering

2.10 Fairness

In order to analyze or prove (even informally) properties of a program written in some conventional sequential programming language one normally makes an implicit *positive speed* assumption. In a statement oriented language the assumption is that statements are executed in a positive speed. More generally, the assumption is that if a point in the sequence of steps executed by a program is reached, then the next step will eventually be executed (if its execution is permitted by the program). More specific assumptions can be made if more information is known about the implementation; but such information is not provided by the definition of a language.

In a language for parallel programming the notion of a sequence of steps is not directly applicable. However, assumptions which are analogous to the positive speed assumption should also be made for analyzing or proving properties of programs. In a process based model each process is generally viewed as a sequence of steps. The corresponding assumption is that each process is executed in a positive speed. In addition to the nondeterminism which is inherent to languages for parallel programming, the corresponding models can also be nondeterminate. In such cases, it may be more difficult to express the analogue to the positive speed assumption. Our model belongs to this latter category.

This discussion leads to the problem of fairness of an implementation of a language. If a language definition does not contain some fairness rules which must be met by every implementation of the language one may not be able to analyze or prove properties of a program which are implementation independent. In general, only weak fairness rules should be included in a language definition in order not to constrain too much implementations of the language. An example of a weak fairness rule is: if a computational activity can proceed it eventually proceeds.

Definitions of most existing languages do not include fairness rules; although limited rules for several constructs may exist. As an example, suppose one writes a program in which two processes P_1 P_2 are activated. Each process consists of a nonterminating loop. In each cycle process P_i prints some number on printer i. If such a program is written in languages such as: PL/1, CONCURRENT PASCAL [BH-75], MODULA [Wi-77b], or communicating sequential processes [Ho-78a]; nothing guarantees the programmer that process P_1 (P_2) will print even one number. Our language defers from other languages in that fairness rules are part of its definition. Using these rules one can easily show that infinitely many numbers are printed on each printer. Act1 [He-79] manifests a similar property by its guarantee of service.

This section gives some rules about the way a computation proceeds in our model. These rules are quite relaxed in order not to constrain too much implementations of the language. The rules try to guarantee two separate requirements. First, that the computation will eventually proceed if possible. Second, that some fairness is provided. Formal treatment of the fairness problem, based on temporal logic, appears in [Pn-79]. It is true that the language contains predicates which allow the user to have some explicit control over the fairness of the execution. However, in our opinion the language should guarantee at least a minimal fairness even if the user does not use the predicates. Fairness is normally discussed in the context of concurrent processes competing for resources. Our model is different from a process based model therefore the notion of fairness is different in the context of EBL. The fairness rules discussed later in this section explicate our notion of fairness.

One must observe that the fairness rules do not rule out the possibility of starvation from every program. Suppose, for example, the heading of event handler H is:

on $E_1 \wedge E_2$

where E_1 and E_2 are single_use event class identifiers that also appear in the event descriptor lists of other event handlers. The fairness rules do not rule out the possibility that no instance of H will ever be activated even though infinitely many times a pair of events (one from each of the event classes E_1 and E_2) exists. The fairness rules guarantee, however, that the scheme for expressing a P operation on a semaphore variable (see chapter 5) is fair (in the sense that if enough V operations are executed then every P operation eventually terminates).

The first fairness rule is:

F0. Once an instance of an event handler is activated, execution of its script eventually terminates.

In fact, F0 should be conditioned on availability of enough computational resources. We shall assume that such qualifications are added to all our fairness rules. The motivation for F0 is that causing each event in the script of an event handler can be executed in finite time, and the number of events specified by the script of an event handler is fixed. Thus, the execution of the script can be completed in finite time.

F0 is not sufficient since an implementation of the language may choose not to activate any event handler instance at all. Such an implementation does not violate F0 but is certainly unacceptable. Before presenting the next rules we define several conditions which may be satisfied by a program. These conditions are then used in the rules themselves.

Let e_i be an event from class E_i which exists at time t_0. Let H be an event handler. Assume C is a clock ticking at some fixed positive finite rate; the time interval defined by two adjacent ticks is the clock's *period*. The various conditions and rules are defined next.

C1. If e_i is not used at any time $t \geq t_0$, then infinitely many periods following t_0 contain points in time at which e_i can be a member of at least one event collection which could activate an instance of some event handler.

C2. E_i is of a multi_use type. If e_i is not used by instances of H at any time $t \geq t_0$, then infinitely many periods following t_0 contain points in time at which e_i can be a member of at least one event collection which could activate an instance of H.

C3. The event descriptor list of H contains only multi_use event class identifiers; there are m event descriptors, and the corresponding event class identifiers are $E_1, \ldots, E_m$. $e_c=\{e_1, \ldots, e_m\}$ is an event collection associated with the above event class identifiers. If e_c does not activate an instance of H at any time $t \geq t_0$, then infinitely many periods following t_0 contain points in time at which e_c could activate an instance of H.

Several fairness rules can be defined based on the previous conditions.

F1. For all e_i, t_0 if condition C1 is satisfied then eventually e_i is used at time $t \geq t_0$.

Rule F1 guarantees that processing of a request (represented by e_i) which can be handled by one event handler or by several event handlers eventually begins.

F2. For all e_i, H, t_0 if condition C2 is satisfied then eventually e_i is used by an instance of H at time $t \geq t_0$.

Rule F2 guarantees that a message (represented by e_i) broadcast to several event handlers is eventually received by all of them.

F3. For all $e_c=\{e_1, \ldots, e_m\}$, H, t_0 if condition C3 is satisfied then eventually e_c activates an instance of H at time $t \geq t_0$.

Let $\mathcal{E}_i(t)$ be a collection of existing events from event class E_i at time t for $i=1, \ldots, m$. Suppose one wants to perform some computation on each element e_c in the cartesian product $\mathcal{E}(t)=\mathcal{E}_1(t) \times \ldots \times \mathcal{E}_m(t)$ which is built as time progresses. Rule F3 guarantees that if $e_c \in \mathcal{E}(t_0)$ then the computation corresponding to e_c eventually begins (possibly before t_0). In other words, F3 guarantees that conduits [Wa-78a] can be correctly implemented in EBL (without using EBL's predicates).

All the fairness rules have the form: if some condition is satisfied then eventually S (something) occurs. This is a very relaxed form and an implementation of the language should try not to delay unreasonably the occurrence of S. However, we do not include the additional requirement in the definition of the language.

2.11 Summary

An overview of EBL has been presented in this chapter. The language is based on events which provide the only control mechanism. Events are explicitly caused by the program and they activate instances of (dynamic) program units called event handlers. The only operation that can be performed by an instance of an event handler is the causing of new events (which can activate more instances of event handlers). Many event handler instances can be executed concurrently. In contrast to event handlers, modules have only static effects; they have no effect at run time. Several fairness rules are part of the language definition; they must be met by every implementation of EBL.

A deeper insight into the language is given in chapter 3. The definition of the language is completed in chapter 4.

3. Relation to Other Mechanisms

The purpose of this chapter is to give a deeper insight into the language before diving in the next chapter into the detailed definition of the language. It starts with a comparison of certain features of our event model with other models (process based models, and message passing models); continues with an analysis of different kinds of modularity in the language; discusses possibilities for parallelism in the language; proceeds with a discussion of synchronization capabilities in the language; and then shows the relation between the language and some other models of computation.

3.1 Relation to Other Models

This section examines why event semantics seem promising as the underlying basis for a parallel programming language for distributed systems. This is done by comparison to other models: message passing models such as the Actor model [He-76] and the mu calculus [Ha-78, Wa-78a], MODULA [Wi-77b] an example of a process based model, and procedure based models such as ALGOL [Na-63].

The message passing models and the event model on one hand, and MODULA on the other hand, are all intended to allow expressing concurrent computations. In the first cases, each activity can unilaterally spawn new activities, thus forming a multilevel tree of concurrent activities. In the latter case, the computation is restricted to cooperating sequential processes, only one of which (the main program) can create new processes. In order to achieve a multilevel tree of concurrent activities, whenever a secondary process (not the main program) wishes to create a subprocess it has to request it from the main program. The communication with the main program is needed in general since a process decision to spawn new subprocesses may be data dependent. Observe, however, that the above limitation of MODULA is not inherent to every process based model; PL/1 for example, does not suffer from such limitations.

The disadvantages in the case of MODULA are additional communication overhead, and the bottleneck created by the need to communicate with one process. For example, the time required to initiate $N=2^n-2$ activities, whose causality relation has the form of a constant depth (n-1) rooted complete binary tree, can be proportional to the height of the tree n-1 (i.e., O(log N)) in the case of the message passing models and the event model (assuming at least N free processors), but must be at least proportional to the number of nodes in the tree (except the root) 2^n-2 (i.e., O(N)) in the case of MODULA. Thus, the event model shares an advantage with the message passing models over MODULA in that they both allow spawning of new activities at a higher rate, therefore expressing a higher degree of concurrency within a computation.

Let us describe the basic computational steps of several models in message passing terms. In a procedure based model, a procedure call involves the activation of a procedure and the subsequent return of a value. This can be viewed as passing a message from the activator to the procedure, followed by passing a message from the procedure to the activator. In this case, the basic computational step consists of two interactions, each involving two instances of program units. In a message passing model the basic computational step is simpler; it only includes the passing of a message from one instance of a program unit to another one. Here, there is one interaction between two instances of program units. In our event model, the basic computational step is even simpler; it only includes the sending of the message. Here, only one instance of a program unit is involved and the interaction is between it and the ether (or between it and the system).

Let us examine how a computation consisting of several nested subcomputations (at levels 1, ... , n) progresses in several models. In a procedure based model, the computation terminates when all the subcomputations at levels 1, ... , n terminate. In a message passing model, starting a subcomputation corresponds to passing a message to an instance of a program unit. The passing of the original message causes passing of messages activating the subcomputations of level 1. The processing of the original message terminates (conceptually) when the messages activating the subcomputations of level 1 are *received* by their targets (and not when all the nested messages are processed).

In the event model, starting a subcomputation is achieved by causing an event. In the processing of the original event, new events for activating the subcomputations of level 1 are caused. The processing of the original event terminates when the messages activating the subcomputations of level 1 are *sent* (broadcast); i.e., possibly prior to the association of messages with their receivers.

Thus, in the event model an instance of a program unit can vanish, relinquishing computational resources, as soon as its messages are sent and before the possible receivers observe them. The price for this early release of computational resources in the event model is paid by the receivers which have to check whether they are interested in the messages or not. The semantics of our language restricts the required checking (by the use of event classes): a program unit need not check all messages but only those from certain classes.

Suppose one wants to perform some computation on each element in the cartesian product $S_1 \times ... \times S_k$. Further, suppose each element of S_i is represented by a message (event) in the class associated with S_i. In our model, an instance of a program unit can specify that any collection of k messages (events), each from a specific class, activates an instance of the program unit. If n messages are broadcast from each of the k classes, totally $k*n$ messages, n^k instances of the program unit are activated, one for each combination of k messages from those classes. In the Actor model, each message activates one instance of a program unit, thus in order to achieve the same effect at least n^k messages are needed. In addition to the higher number of required messages, the programmer has to explicitly write an appropriate algorithm in the Actor model, in contrast to our model. The *conduits* of the mu calculus [Wa-78a] allow expressing the above behavior easily; however, the number of messages sent is at least $k*n+n^k$. (Equality can be reached by extending the conduit construct to handle k classes of messages, and not only two classes as defined in [Wa-78a].) Note that the smaller number of messages required in our model to activate n^k instances of the program unit is only a matter of level of abstraction. At a lower level of abstraction (the language implementation level) some signal is needed to trigger the activation of each instance of the program unit.

The ability of an event object to carry event class identifiers as parameters provides for a special programming style which has lately become popular, the use of *continuations* [St-74, He-76, Ha-78, Wa-78a]. An event handler can be passed a parameter which specifies what to cause when done, namely, instances of which event handlers to activate next. Continuations are used in several places in this thesis; some of them are: the factorial module in the example in chapter 2, the implementation scheme for procedures (see chapter 5), and the airline reservation system (see chapter 6).

Various message passing schemes can be built on top of EBL. The fact that an event has parameters associated with it allows it to behave as a *message carrier*; in addition to the activation of some processes, the event delivers them the carried message. By using **multi_use** events, broadcast modes can be implemented, since causing an event actually broadcasts some message, the event object, "To Whom It May Concern".

The event model allows broadcasting a message that can be received by any (one) program unit which finds it interesting out of a group of n program units; leaving the task of selecting the program unit to the system, when more than one program unit is interested in the message. In order to achieve the same effect in the Actor model the programmer has to devise an appropriate algorithm. The same algorithm to be used for the implementation of our language can be employed for this purpose. The identity of the n program units must be known to the algorithm in case of the Actor model, whereas no such requirement exists in our model.

The ability of an instance of a program unit in the event model to broadcast a message to several program units has no analogue in the Actor model or in the mu calculus. So is the ability to broadcast a message without knowing what program units will receive it, how many instances of program units will receive it, and when will it be received. As a special case, the event model allows passing a message to one instance of a program unit, as is done in the above models. Pattern directed invocation languages such as ETHER [Ko-79] manifest such capabilities; the relation to our model is investigated in section 3.7.

The previous paragraphs discussed some of the differences in the expressive power of the various models. It seems that the expressive power of the event model is at least equal to that of the message passing models (with the exception of the basically global environment used in the event model in contrast to the local (nested) environments used in the message passing models (except for ETHER [Ko-79])). This of course assumes that the event model allows the same kinds of messages and the same capabilities for processing a message as the message passing models.

3.2 Mutable Objects

Some mechanism for achieving the effect of mutable objects (or causing side effects) seems needed in order to obtain a language with high expressive power. Such a capability is especially important in a language for parallel programming where concurrent interacting activities are dynamically spawned, synchronized, and joined. One may argue that side effects can be simulated in applicative languages; e.g., in the lambda calculus it can be done by passing the "current state" of all the machines in the system to every lambda expression [He-76]. However, such a programming style is unnatural, and its implementation on a distributed system is bound to be inefficient.

Mechanisms for achieving the effect of mutable objects can be classified as *destructive mechanisms*, and *additive mechanisms*. In the former case, modifying the state of an object permanently destroys some (or all) information about its previous states. In the latter case, modifying the state of an object only adds new information to the information about its previous state; i.e., information is never lost (e.g., a list on which the only allowed operations are **append** and **read** element i, or eventcounts [Re-77a]). The most common mechanism (for this purpose) in programming languages is the use of variables and *assignment statement*; this is obviously a destructive mechanism. A similar concept in the context of message passing is the *cell* [Gr-75] which is used in the Actor model; this is also a destructive mechanism. As was pointed out in [Ha-78], the use of cells should be avoided whenever possible since it complicates the semantics of a language. Cells, however, (and destructive mechanisms, in general) save space by destroying past information. The mu calculus incorporates the *conduit* construct [Wa-78a] which only allows a confined kind of mutability. It is a compromise between allowing and forbidding the use of mutable objects; this is an additive mechanism.

In EBL there are no mutable objects; therefore, in light of the above discussion, this could limit the expressive power of the language. However, the effect of mutable objects can be achieved due to:

1. The ability to create permanent objects belonging to specific classes (cause **multi_use** events), and to read the latest object from an event class; this is an additive mechanism. This approach is pursued in the readers writers example in chapter 6.
2. The ability of a program unit to delete several temporary objects belonging to specific classes (use **single_use** events), and to create new objects from the same classes; this is a destructive mechanism. This approach is pursued in chapter 5. This ability adds to the expressive power of the language but also complicates its semantics.

3.3 Modularity Issues

Modularity is manifested in several forms in the language; we use the same name in discussing all of them. The first source of modularity stems from the semantics of modules. The mechanism of exporting - importing of identifiers provides for hiding and sharing of identifiers between program units in a controlled way. For example, in Figure 2.1 F is known in both modules but not outside the outer module. Modules protect event class identifiers against undesired uses (in the sense of section 2.2). This protection is especially important in the case of **single_use** events, which may vanish if an unauthorized event handler uses them. For example, in Figure 2.1 F is protected by the factorial module; similarly, sum_f is protected by the factorial sum module.

Modules allow creating encapsulated program units and abstract data types. One can write a module, specify its behavior in terms of the relation between events it uses and events it causes, and then use it without paying attention to the way it is implemented; i.e., in a bottom up manner. For example, in Figure 2.1 one can begin by writing the factorial module. The behavior of this module can be specified as follows: Each event of the form F(n,1,C,t) causes an event of the form C(r,t) where r=factorial(n) if $n \geq 0$, else 1. When this module is then used one does not have to know whether factorial is computed iteratively, recursively, or by table lookup. Similarly, the behavior of the factorial sum module can be specified as follows: each event of the form sum_f(i,j) causes an event of the form print(k); k=factorial(i)+factorial(j) where factorial(n) is 1 if n<0.

The above properties of modules are also desirable for a top down design of a program. Like in other languages, a program can be designed by step-wise refinements

[Di-72]. At each step a module contains only trivial event handlers that activate inner, still non-existing, event handlers and returns (causes) the correct result events when the inner event handlers return (cause) their result events. The next refinement consists of defining inner modules, in which inner event handlers implement the desired behavior of the previous step, again to a certain level of abstraction. This process continues until the innermost event handlers which actually perform the necessary computation are written.

As an example consider the factorial sum problem of chapter 2. Suppose the activating event is sum_fact (instead of sum_f). In the first step, the program can consist of the following module:

```
module
        export: sum_fact ;
        import: print ;
        sum_fact: event (int, int) ;
        sum_ready: event (int) ;

        on sum_fact (i, j: int)
                par_cause sum_f (i, j)
        end ;

        on sum_ready (i: int)
                par_cause print (i)
        end ;
end ;
```

In the second refinement the subprogram of Figure 2.1 except the factorial module can be added within the above module; print(i+j) is replaced by sum_ready(i+j), and sum_ready is added to the import list. The last refinement involves writing the factorial module within the factorial sum module.

If programs are written in the way described above the intermediate steps need not be deleted from the program at each refinement step; they can remain there. A clever compiler can eliminate from the program the causing of intermediate events which constitute the hand-shaking between event handlers of nested modules, and thus the efficiency of a program is not affected by the top down design. An obvious advantage of this approach is that all the design steps remain in the final program text. The text may be longer than the one obtained if intermediate steps are deleted. However, the program is easier to understand and maintain since it is hierarchically built.

Another kind of modularity is provided by the language (a similar property is described in [Ko-76]). It stems from several reasons. First, the order of event handlers or modules within the module containing them, or within the program (if they are not contained in any module), has no effect on the meaning of the program. Second, an event class identifier can appear in the headings of more than one event handler. If an event from such an event class is of a **single_use** type it activates at most one event handler instance; if it is of a **multi_use** type it may activate instances of more than one event handler. Thus, the algorithm for handling an event can be distributed in the program text. This feature can be useful, as discussed next.

A real time program for controlling a physical system, such as a car, can be constructed as a collection of modules that reflects the structure of the physical system; each program module will correspond to some physical module. Thus, a program for controlling a car may contain program modules that correspond to the ignition, the oil, the brake subsystems, etc. Each such program module encapsulates all the necessary operations related to the corresponding physical module. An external event of the physical system that influences several physical modules, will have several associated event handlers distributed in the corresponding program modules. Similarly, a regular program event can activate several event handlers distributed in the program. For, example, an event from the class *check* can initiate checks in the program modules that correspond to the ignition, the oil, the brake subsystems, etc. On replacement of one of the physical modules, only the corresponding program module has to be modified.

3.4 Parallelism in the Language

Parallelism in an EBL program is manifested at several levels. At the top level, several instances of event handlers can be executed concurrently; each can cause new events which activate more instances of event handlers. A previous section demonstrates that the rate of spawning new activities in this manner is high, e.g., in comparison with MODULA. At a lower level, when an instance of a parallel handler is activated the events specified in its script can be caused concurrently. Parallelism at the above two levels is explicit in the language definition. The following analysis shows that a great deal of parallelism can be exploited at a lower level; this applies not only to a parallel handler but also to a sequential handler.

Once an instance of an event handler is activated, all the data needed for evaluating the parameters of the events it causes are available. In fact, the value of each parameter is a function of the formal parameters in the event handler heading and of some identifiers (event class identifiers, and tag identifiers). The values associated with the identifiers in an expression defining a parameter of an event to be caused are fixed and cannot be modified during the whole computation. Furthermore, evaluating those functions causes no side effects. Therefore, when an instance of an event handler is activated all event parameters in its script can be evaluated in any order or concurrently (for both types of event handlers).

Earlier it was argued that the difference between a sequential handler and a parallel handler is that the events caused by an instance of a sequential handler are ordered by the precedes relation, whereas those caused by a parallel handler are not. The above discussion shows that the difference is basically in the order in which the event objects containing the computed parameters are inserted into the conceptual event lists. In the former case these insertions are ordered whereas in the latter case they are not. Another consequence of the previous discussion is that once an instance of an event handler is activated its script can be executed in a finite time. All the data needed for the execution are available; therefore, there is no a priori need to suspend it during the execution.

3.5 Synchronization Capabilities

This section contains a comparison of synchronization capabilities in EBL with other mechanisms such as semaphores and monitors. As is shown in chapter 5, general semaphore variables [Di-68a, Di-68b] can be implemented as event class identifiers of a single_use recurrent type. (For achieving mutual exclusion, single_use non_recurrent events are sufficient.) In fact, *multiple P operations* [Di-71, Pa-71] can be easily expressed in the language, as is shown in that chapter. single_use recurrent events are more powerful than the general semaphore, since they may have parameters that can be used to carry some information in addition to their behavior as synchronization primitives. The counter associated with a semaphore variable is implicitly represented by the number of occurred events which have not been used so far; i.e., the number of existing events from the event class representing the semaphore.

When a V operation is executed on a semaphore variable, one of the *currently waiting* processes is arbitrarily selected and resumed. EBL's analogue to a process is an instance of an event handler; the notion of a waiting process is not applicable since there is no construct similar to a wait statement (although such behavior can be easily modeled). When the analogue to a V operation is executed in EBL, it guarantees the resumption of some "process" executing the analogue to a P operation. The selection of this process is even less specified than in the case of semaphores since a process which is currently not waiting (but will start waiting sometime in the future) may be selected. EBL's predicates however, provide the ability to explicitly control the extent of fairness of an algorithm, as will be shown in chapter 6.

The obvious difference between EBL's events and the monitor construct [BH-74, Ho-74], is that the latter is more structured. The mutual exclusion which is implicit in the monitor must be explicitly expressed in EBL because EBL's events are primitive objects which are closer to the semaphores than to the monitor.

The implicit mutual exclusion is not primitive enough; for example, if a process which entered the monitor decides to wait for some condition (which depends on the monitor's state variables) to become true it has to leave the monitor and reenter in the future. In order to prevent this form of busy waiting the monitor also allows explicit synchronization operations on condition variables. It might be interesting to compare the monitor condition variables with single_use recurrent events. Signaling a *condition* has no effect if no process is waiting for it; i.e., no memory is associated with monitor conditions.

However, in EBL, a single_use recurrent event is remembered until it is used. As described in [Wi-77c], a deadlock may occur when trying to synchronize parallel processes using the monitor condition variables (instead of using semaphores) because of the lack of memory. This kind of deadlock cannot occur if single_use recurrent events are employed since such events are not forgotten before they are used; i.e., before they have some effect.

The monitor construct also allows the association of a priority with a waiting process by the use of *scheduled waits*. A comparison of this feature with EBL's mechanisms is given in chapter 6 through the disk head scheduler example.

The mechanism used for synchronization in the Actor model is the *primitive serializer* [He-79]. In the Actor model two kinds of actors can be created: an *unserialized actor* and a *serialized actor*. An actor of the first kind can receive and process several messages concurrently. An actor of the latter kind processes messages one at a time, i.e., it serializes them. The serialized actor becomes locked when it starts processing a message; it becomes unlocked when processing of the message has completed. The serialized actor is in fact an adaptation of the monitor to the Actor model.

An advantage of the serialized actor over the monitor is that in the first case, during the processing of a request operations can be carried out concurrently, whereas in the latter case, when a process enters a monitor's procedure its execution proceeds sequentially. EBL shares the same advantage of the serialized actor over the monitor in that concurrency can be employed during the processing of a request.

Both structured constructs, the monitor and the serialized actor, can be implemented using the unstructured constructs semaphores and cells (variables). Since the above unstructured constructs can be expressed in EBL, then so are the monitor and the serialized actor. The state of a monitor is kept in local variables. The state of a serialized actor is kept in its behavior (another actor); this behavior can be modified by the analogue of an assignment statement, a *become* command. The event handlers implementing a monitor do not have any state information associated with them. This demonstrates the uniqueness of our events: the event used to synchronize accesses to the monitor can carry as parameters the state of the monitor.

3.6 CSP, Events, and Actors

This section compares certain characteristics of communicating sequential processes (denoted here by CSP) [Ho-78a] with EBL. In addition, some of the characteristics are also compared with the Actor model since such comparisons give a deeper insight. Since CSP is a process based model some of the points discussed earlier in this chapter can be applied. We shall not repeat the discussion here. CSP is a static language in several respects. For a given program the maximum number of concurrent processes is bounded. Similar limitations do not exist in EBL or in the Actor model. A given subprogram which recursively computes factorial(n) can do so only for n smaller than some given limit. In CSP the set of processes with which a given process can communicate is fixed and can be found directly from the program text; i.e., there are no process type variables (in contrast to local variables of a process which exist in CSP) and no process type parameters. EBL and the Actor model are more dynamic in nature. The first allows using event class identifiers as parameters of events, and the second allows specifying continuations; both features are similar. The feature of event class identifiers as parameters does not complicate the implementation of EBL and increases its modularity and expressive power. This feature makes programs harder to analyze, but this is a secondary issue in our opinion.

CSP does not have the features of process type variables or process type parameters. This limitation together with the property that each process must explicitly name the destined process in each message it sends do not allow one to implement procedures and use them in a modular manner. In particular one cannot write a procedure named S, then add to the program a process having a new name X which activates S; knowledge of X must be incorporated in S. In EBL procedures can be expressed in several ways. One makes use of event class identifiers as parameters, thus allowing continuations (see chapter 5); this form does not exist in CSP. The other makes use of tag identifiers. Upon termination of a procedure it can cause an event from a fixed event class. A parameter of this event can be a tag supplied to the procedure at call time by the activating event. This scheme can be viewed as a generalization of the use of an array of (not necessarily identical) processes P(i) calling the same procedure in CSP; our tags are dynamic in contrast to the static nature of indexes.

A unique characteristic of CSP which has no analogue in EBL or in the Actor model is that a receiver must explicitly name the possible senders. In CSP the sending of a message is synchronized with the receiving of the message. This unique feature makes synchronization of processes an easy task; however, it has several drawbacks which are not shared by EBL or by the Actor model. The sender cannot proceed until the receiver is ready to accept the message. Messages sent from one process to another are not buffered; both EBL and the Actor model have unlimited buffering capability. If two processes X and Y exchange messages and process X sends a nonmatching message to process Y then a deadlock occurs in CSP. In the Actor model Y can complain about the message. In EBL the chances of a nonmatching message are greatly reduced since the language is strongly typed. A nonmatching message of the correct type can be dealt with in any desired way; this can be achieved by means of an event handler whose instances are activated when such nonmatching messages arrive.

Another important difference between EBL and CSP concerns fairness. In this respect EBL is distinguished from other languages as well, as noted in chapter 2. Consider the following program (adopted from [Ho-78a]):

```
stop, terminated: event ;
continue: event (int) ;

on program_start
   par_cause continue (0) ; stop        { initiate }
end ;

on continue (n: int) ∧ stop
   par_cause terminated                 { terminate }
end ;

on continue (n: int)
   par_cause continue (n+1)             { iterate }
end ;
```

Our fairness rules imply that this program eventually terminates. Hoare, on the other hand, does not require from implementation of CSP to be fair; thus, the analogous program in CSP may loop forever. A similar program can be written in Act1. The first event handler in the above program can be replaced by one actor (ignoring the fact that the two messages sent by the event handler are not ordered). The last two event handlers can be replaced by a serialized actor accepting messages of two kinds: *continue* and *stop*. The obtained Act1 program is not equivalent to the above program but it is sufficient for the following discussion. The Actor model (correctly) does not make any assumption on the order in which two messages (e.g., *continue* and *stop*) sent from one actor to the same target reach their destination. However, since a serialized actor processes messages in the order in which they are received, the corresponding Act1 program stops iterating after the *stop* message is received.

Suppose knowledge that the event from class *stop* has occurred is brought to knowledge of the second event handler (or to its implementation) at the same time a *stop* message arrives to the serialized actor (assuming implementations of the EBL program and the Act1 program run concurrently on different systems). The Act1 program terminates in the next iteration of the serialized actor. EBL, however, only guarantees that the program eventually terminates. CSP does not guarantee termination at all; the process trying to send the *stop* message may wait forever if the destined process decides not to accept the message.

3.7 Relation to Pattern Directed Invocation Languages

Pattern directed invocation languages are used for problem solving in artificial intelligence. Their development started by Hewitt [He-69] and is still in progress. Out of these languages, the one which is most reminiscent of ours is ETHER [Ko-79]. (Its existence was brought to our attention by Hewitt toward the completion of our research, in the spring of 1979.) A subset of ETHER resembles an earlier version of EBL (which has developed since). ETHER's *sprites* and *assertions* are analogous to EBL's event handlers and events respectively. More precisely, an ETHER assertion is analogous to a multi_use

non_recurrent event. The lack of the analogues to EBL's other event types in ETHER suggests that certain computations which can be expressed in EBL cannot be expressed in ETHER. Our **single_use** events support modeling of mutable objects and synchronization primitives; these cannot be modeled in ETHER.

In ETHER a sprite can contain other sprites; the inner sprites are exposed after the containing sprite is activated. An earlier version of EBL also allowed nested event handlers but this feature was eliminated from the language in favor of multiple event descriptors in an event handler heading. The expressive power in both cases seems similar when only **multi_use** events are employed. However, when **single_use** events are allowed the feature of multiple event descriptors increases the expressive power; e.g., a multiple P operation can be easily expressed in this case. Another prime factor in our decision to eliminate nested event handlers is ease of implementation. In the current version of EBL the number of event handlers exposed at any point in time is fixed and they all operate in the same environment (except the static effect of modules). When nested event handlers are allowed, the number of exposed event handlers vary with time and they do not operate in one environment, but rather, nested environments are needed.

The mechanism by which a sprite decides whether an assertion matches it is comparison of the assertion to a specific pattern. This is less powerful than the mechanism used in EBL which allows a generai boolean expression (and predicates) for this purpose. ETHER is based on broadcast and pattern matching mechanisms but it relies on other control mechanisms as well. For example, conditional constructs are allowed within the body of a sprite. As another example, ETHER allows a sprite to explicitly abort some activity in progress by using a special construct (*stifle*). No special construct is needed in order to achieve this effect in EBL. A subcomputation can be aborted, and even be resumed, using the basic event mechanism (by including the predicates **exist** or **none** in event handler headings).

3.8 Relation to Production Systems

The structure of an EBL program is reminiscent of a collection of productions in a production system [Wi-77a]. An event handler is the analogue of a production. The event handler heading corresponds to the *situation recognition part*, and the event handler script to the *action part*. A production whose situation recognition part is a conjunction of conditions can be easily expressed in EBL since an event descriptor list is a conjunction of event descriptors. A production whose situation recognition part is a *disjunction* of *n* conditions can be expressed, for example, as *n* event handlers with identical scripts. The common script causes a **multi_use non_recurrent** event which activates the action part of the production (another event handler).

In a deduction oriented production system [Wi-77a] one normally begins with a collection of facts and tries to reach a goal (in forward chaining mode). A fact may be used several times in the process of moving toward the goal; in addition, a fact may be established more than once in that process. Therefore, our **multi_use non_recurrent** events are appropriate for such systems. In order to solve several unrelated instances of a problem concurrently one can simply use tags to obtain the desired behavior. In other types of production systems **single_use** events can be employed. For example, in a system for manipulating objects, once an action in a sequence of actions to be performed is taken there is no need to remember it.

The fundamental difference between production systems and our model is that the first is a sequential model whereas the second is a parallel model. Productions are invoked one at a time. If several productions can be invoked at some point then the next production is selected according to priority or some heuristic algorithm. Such a behavior can be modeled in EBL but better alternatives are available in EBL. In a deduction oriented system, for example, all possible paths can be pursued concurrently. (ETHER [Ko-79] is based on this idea.)

3.9 Relation to Petri Nets

The modeling power of Petri nets is less than that of Turing machines [Pe-77]. Therefore, one cannot expect that a program in EBL, which is clearly a universal language, could be modeled accurately by a Petri net. However, if one agrees not to model data dependency in a program, in particular to ignore all where clauses (i.e., to assume that they are true for each event collection), then a Petri net can be created for many programs (or subprograms) in EBL. A Petri net can be created for every EBL program satisfying the following conditions:

P1. Each event handler H contains exactly one event descriptor in its event descriptor list, or at least one of its event descriptors is associated with an event class identifier of a **single_use** type.

P2. There are no event class identifiers of **non_recurrent** types.

P3. There are no formal parameters of type event.

In order to show how the Petri net is created we begin with a program in which in addition to the above conditions all event class identifiers are of **single_use** types.

The basic method is to associate one place with every event class identifier of a **single_use recurrent** type, and one transition with every event handler. The script of a parallel handler is modeled by arcs going from the transition associated with the event handler to the places associated with the classes of the caused events. In case of a sequential handler, the script is modeled by a chain consisting of arcs places and transitions starting at the transition associated with the event handler. The chain reflects the order in which the events are caused. An additional arc goes from each transition in the chain to the place associated with the class of the caused event. Figure 3.1 depicts the Petri net corresponding to the following subprogram, in which all event class identifiers are of a **single_use recurrent** type.

```
on P1 (i: int) ∧ P2 (j: int) where i>j
   par_cause
         P1 (i-1) ; P3 (j+1)
end ;

on P1 (i: int) ∧ P3 (j: int) where i=j
   seq_cause
         P1 (i+1) ; P2 (j+2)
end ;
```

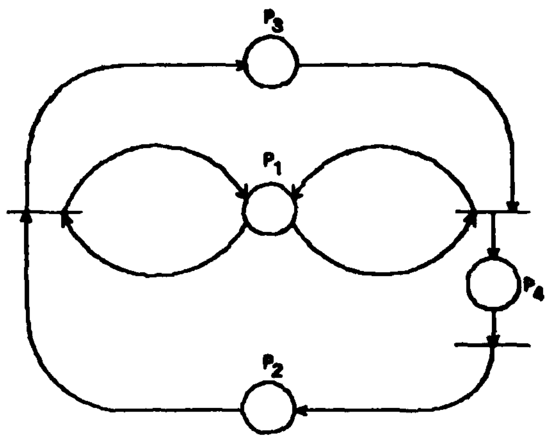


Figure 3.1 Petri net representation of a subprogram

The above scheme may require multiple arcs connecting a place and a transition (in the same direction). Such generalized Petri nets are equivalent to ordinary Petri nets [Pe-77].

The modeling of multi_use events requires some sophistication but is still not complicated if conditions P1-P3 are satisfied. Suppose P_i is a place associated with an event class identifier of a multi_use recurrent type appearing in the headings of n event handlers $H_1, \dots, H_n$; n additional places $P_{i1}, \dots, P_{in}$ are created. The input arc to the transition associated with H_j comes from P_{ij} instead of from P_i; an output arc goes from that transition to P_{ij}. There is only one outgoing arc from P_i; it goes to a transition whose n outgoing arcs go to $P_{i1}, \dots, P_{in}$. This solution simply copies the tokens of P_i to n places P_{ij}, each associated with one event handler, and recreates them in each P_{ij} whenever they are swallowed. Figure 3.2 depicts the Petri net obtained if in the previous example P1 is changed to be of a multi_use recurrent type.

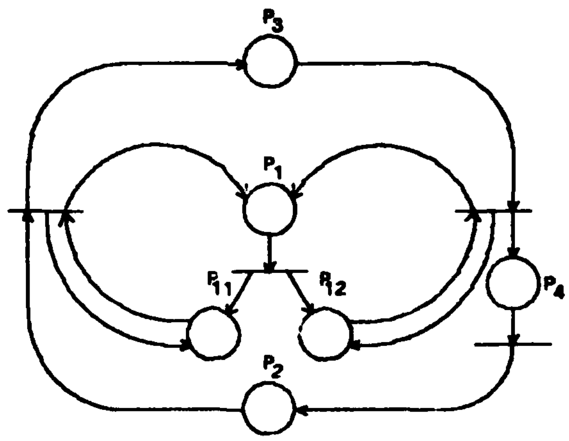


Figure 3.2 Modeling multi_use recurrent events

The scheme presented above does not reflect the existence of event parameters. Since a non_recurrent event is caused only if an identical event (an event from the same event class having the same parameters) does not currently exist, this scheme is not appropriate. The scheme can be extended in order to model non_recurrent events without parameters but we shall not do so. Ordinary Petri nets are sufficient in this case; in particular, there is no need for zero testing [Pe-77].

Another difficulty is the use of formal parameters of type event. The problem is that the class of an event whose class designator is a formal parameter cannot be determined from the script of the event handler and is not fixed. As an approximation, one can find the set S of all possible event class identifiers to which such a formal parameter can be bound. S can be found by a simple algorithm based on transitive closure. The part of the Petri net corresponding to the event having the formal parameter class designator will simply add a token to one of the places corresponding to the elements of S.

The other direction is more successful: each Petri net can be easily expressed as an EBL program. A single_use recurrent event class identifier (without parameters) is associated with each place in the net. Each transition is described by a parallel handler whose script causes events from the appropriate classes. The following program represents the Petri net of Figure 3.1:

```
P1, P2, P3, P4: event ;

on P1 ∧ P2
   par_cause P1 ; P3
end ;

on P1 ∧ P3
   par_cause P1 ; P4
end ;

on P4
   par_cause P2
end ;
```

The initial marking of the Petri net can be easily programmed. One way is to cause events corresponding to the initial tokens in one or more event handlers activated by the event from the class **program_start**. In this case, **single_use recurrent** events without parameters and parallel handlers without where clauses are sufficient to describe any Petri net. Another approach is first to cause the above events and then to *enable* the rest of the event handlers by causing an event from the class *start*.

There are two ways in which an event handler H can check the *start* condition. In the first way, the where clause: **where exist** (*start*) is added to the heading of H; here, *start* can be of a **single_use** or a **multi_use**. In the second way, an event descriptor of the form: ∧ *start* is added to the event descriptor list of H. Here, *start* can be of a **multi_use** or a **single_use** type; in the latter case, an instance of H should cause another event from the class *start* in order to enable other event handlers (since it uses one when it is activated).

3.10 Relation to Databases

An analogy can be drawn between the language and data manipulation languages for relational databases. An event class is analogous to a relation, and each event to a tuple in a relation. The structure of an event handler (in particular its heading) brings to mind a query written in a data manipulation language such as SEQUEL [Ch-74a]. They both define a nonprocedural operation to be performed by the system. The difference is that once a query is initiated, it runs to completion and then vanishes while the database continues to change. An event handler, on the other hand, is immortal; it stays alive as long as there is a possibility of change in any of the event classes (and when no such change can happen, the whole program terminates). It is analogous to a query whose execution never terminates. In order to model a database in EBL some scheme for achieving mutable objects is needed. Such schemes are presented in chapter 5, and in the readers writers problem in chapter 6.

3.11 Summary

This chapter has exposed some of the important properties of EBL by comparisons to other mechanisms. The language constructs are primitive; nevertheless, the capability of hierarchical program design is provided by modules. Other forms of modularity which are provided by the language have been discussed. A great deal of parallelism can be exploited in an EBL program. New activities can be easily spawned, synchronized, and joined. P operations (and multiple P operations) and V operations on semaphores, monitors, and serialized actors can be expressed in EBL.

Our model shares some of its characteristics with other models of computation; however, important differences exist in each case. This chapter developed the relation between the language and other models such as: message passing models, communication sequential processes, pattern directed invocation languages, production systems, Petri nets, and databases.

4. Definition of the Language

This chapter completes the definition of the language, whose main concepts have already been defined in the previous chapters. It contains a definition of the syntax and semantics of EBL. A formal definition of the syntax is contained in appendix A; it should serve as a convenient reference, and we have not included the formal definition here to avoid redundancy. Appendix A is crdered according to the sections of this chapter, and we suggest that before (or while) reading each section of this chapter, the reader will examine the syntactic structure from the corresponding section in appendix A. The following conventions are used:

1. The syntax is described in an extended Backus-Naur form.
2. A construct enclosed by the braces $\{\ldots\}_m^n$ must be repeated at least m times and at most n times. The default values are m=0 and $n=\infty$.
3. Keywords of the language are printed as **bold face** characters in this thesis.

In contrast to the meta-language braces $\{\ldots\}$, EBL contains the braces **{...}** which are used to indicate the beginning and the end of a comment. Comments can be inserted between any two symbols in the program and have no effect on the meaning of the program; comments may be nested. The chapter starts with the basic constructs; proceeds with the definitions of types, declarations, expressions; and ends with event handlers, modules and programs.

4.1 Identifiers and Numbers

Identifiers serve to denote objects in the language. The objects are formal parameters, event classes, tags, and types. The association of identifiers and objects must be unique within the scope of declaration of the identifiers. Scope rules are given in the section on event handlers and the section on modules. In various places in this thesis subscripts are used within identifiers (e.g., E_i) although this is not allowed by the syntax. The *character set* is an ordered set of all legal characters in EBL. It contains all *digits*, *letters*, and *special characters* in the order in which they are listed in appendix A.

Examples of *identifiers*:

n	{ formal parameter identifier }
sum_f	{ event class identifier }
t	{ tag identifier }

4.2 Constants

Constants appear in expressions. Each constant has a certain type associated with it, as explained in the following section.

Examples of *constants*:

-50	{ integer constant }
true	{ boolean constant }
'a	{ character constant }

4.3 Types and Type Identifiers

Every constant, identifier, or expression has a *type* associated with it. The type determines the set of values that an object of that type can assume, as well as its semantics. The *basic types*: **int**, **bool** and **char** have the conventional meanings. The values associated with objects of type **tag** are obtained from the global *tag set*, as explained in chapter 2. An *event type* represents a group of events. Each event has a fixed number of parameters associated with it, each of a distinct type determined by the corresponding type in the type list of the event type. An event type contains two optional prefixes. The default prefixes are: **single_use**, and **recurrent**. A type can be denoted by a type identifier, as explained in the next subsection.

Examples of *types*:

int	
event (tag, int)	{ **single_use recurrent** by default }
multi_use event (bool, tag)	{ **recurrent** by default }
cont	{ type identifier }

4.3.1 Type Identifier Definition

A type identifier is equated with a list of one or more types. It can be used as a shorthand notation, or as a means for type abstraction. It should be emphasized that if a type identifier is used at any level of nesting within the module in which it has been defined, then its structural details are known at that point; otherwise (outside the module), only its name may be known. The ability to denote a list of known types by a single type identifier allows one to bind a single formal parameter of that type to a list of actual event parameters, when no computation needs to be performed on them. An example of this feature is given in the section on event handlers.

Examples of *type identifier definitions*:

```
        cont, t1 == single_use recurrent event (int, tag) ;
                                        { both cont and t1 represent the same type }

        t2 == int, tag ;                { t2 represents a list of types }

        t3 == single_use recurrent event (t2) ;
                    { t3 is identical to t1 only if the structural details of t2 are known }
```

4.4 Declarations

The term *declarations* actually refers to both declarations and definitions. Type identifier definition has already been described. The declarations of event handlers and modules, which are more complicated, are described later in the corresponding sections; event class identifier declarations and tag identifier declarations are described next.

4.4.1 Event Class Identifier Declaration

An *event class declaration* associates *event class identifiers* with an event type, or with a type identifier whose structure is known as equivalent to an event type. Each event class identifier denotes a class of distinct events of the event type. An event from the event class is determined by the event class identifier and the values of its parameters. Before execution of the program is started, no event from the declared event class has occurred. Events are considered to have *occurred* after they are caused either explicitly by execution of the script of an instance of some event handler, or after the system has caused them. The system can cause either system events or events from a class whose event class identifier appears as a parameter of a system event. This happens, for example, after some system activity has been completed, or when some external signal has been detected, depending on the semantics of the particular system event.

Examples of *event class declarations*:

```
sum_f: single_use recurrent event (int, int) ;
sum1, sum2: cont ;              { cont is known to be an event type }
```

4.4.2 Tag Identifier Declaration

The semantics and use of tags have been described in chapter 2. Note that the value associated with a tag identifier cannot be modified. It can be copied as a parameter of several events; it can be compared with those of other tag identifiers or formal parameters of type **tag**, but there are no other operations that can be done on objects of that type.

Examples of *tag identifier declarations*:

t1, t2: **tag** ; { assume values: α, and β respectively }

4.5 Expressions

Expressions in EBL are rather conventional. They consist of *operands* (constants or identifiers) and *operators*. An operator operates on *arguments* (operands or subexpressions) whose types must be consistent with the operator. Operator precedence is determined from the syntax of the expression. A sequence of operators with equal precedence is executed from left to right. Some of the operators in the language are called *built-in functions*; these are in general more complex operators, but they have nothing to do with conventional functions. Each expression has a type associated with it; this type is either a basic type (**int**, **bool**, or **char**), or a non-basic type; the expressions are named accordingly.

4.5.1 Integer Expression

The operators in an *integer expression* are + (plus), - (minus), * (multiply), / (divide), **div** and **mod**. They operate on arguments of type **int** and yield values of type **int**. The operator / performs division with truncated fraction. In case of the operators **div** and **mod** the divisor must be positive. The operator **div** yields the largest integer not greater than the value of the mathematical division. The other operators have their conventional meanings. The built-in functions are:

1. **abs** which operates on an argument of type **int** and yields a value of type **int**, which equals the absolute value of the given argument.
2. **ord** which operates on an argument of type **char** and returns the ordinal number of the argument value in the character set.

Examples of *integer expressions*:

```
n-1
-(r1 + r2) / (i div j) + abs (k)
```

4.5.2 Boolean Expression

The constants of type **bool** are **true** and **false**. The operators in a *boolean expression* are the *boolean operators*: **not** , **and** , **or**, and **xor**; and the *relational operators*: <, <=, =, <>, >=, >. The former have their conventional meanings and operate on arguments of type **bool** yielding a boolean value. Relational operators yield a boolean value but can operate on arguments which are not necessarily of type **bool**. The operators = (equal) and <> (not equal) can operate on any two arguments of the same type. The values of the arguments are compared and the resulting value is determined accordingly. The meaning of equality is determined as follows:

1. The result of comparing two expressions of a basic type is self explanatory.
2. The result of comparing two identifiers of type **tag** by the operator = is **true**, if they have the same abstract value, from the global tag set, associated with each of them. In the case of two distinct identifiers, this may happen only if at least one of the arguments is a formal parameter of type **tag**.
3. Two identifiers of type event are equal if they represent the same event class (a formal parameter of type event represents the class of events to whose event class identifier it is bound).
4. The result of comparing two formal parameters of a type denoted by a type identifier that represents a list of types is determined as follows: The two arguments are considered equal if each of their corresponding subcomponents are equal according to these definitions.
5. Two identifiers of types whose structural details are not known at that point in the program must be formal parameters, they can be compared with each other only if they are of the same type (denoted by a type identifier). For the comparison the structural details of the common type are made known and the result of the comparison is determined according to the above definitions. Note that comparison of objects whose common type is not fully known at some point in a program is normally done by using a special equality operator which is defined at the same program module in which the common type is defined (see for example MODULA [WI-77b]). The more liberal use of the standard comparison operators (= and <>) in EBL is motivated by the relatively high frequency of

comparisons in programs.

The operators <, <=, >=, and > operate on two values of type **int**.

Examples of *boolean expressions*:

c <= n **and not** b	
t1 = t2	{ comparison of *tags* }
b1 <> b2	{ when both are formal parameters of a type defined as: t2 == **int**, **tag** }

4.5.3 Character Expresssion

A *character expression* can be a constant (a character preceded by a quote), a formal parameter of type **char**, or an activation of the built-in function **chr**. The built-in function operates on an argument of type **int** and returns the character whose ordinal number in the character set is equal to the value of the argument (if such a character exists).

Examples of *character expressions*:

'a	
formal_char	{ formal parameter of type **char** }

4.5.4 Non-Basic Expression

A non-basic expression is an expression whose type is not a basic type. The common characteristic of all non-basic expressions is that they contain no operators. They can be subdivided as follows:

1. The expression is of type **event**. In this case, it can be either an event class identifier, or a formal parameter of the correct event type. In both cases, the value of the expression is the name of the denoted event class identifier.

2. The expression is of type **tag**. In this case, the expression is either a tag identifier, or a formal parameter of type **tag**. In both cases, the value of the expression is the unique value, from the tag set, associated with the identifier in the expression.

3. The expression is a formal parameter of a type denoted by a type identifier that represents either a list of types, or a type whose structural details are not known at that point in the program. In both cases, the value of the expression is the value to which the formal parameter is bound.

Examples of *non-basic expressions*:

c	{ formal parameter of some event type }
t	{ tag identifier }
rec	{ formal parameter of a type defined as: record == **int**, **int**, **bool** }

4.6 Event Handlers

The event handler concept has been explained at some length in chapter 2. Here, some additional details are given. When an instance of an event handler is activated, first its local identifiers are brought into existence: its formal parameters are bound to the corresponding actual parameters of the activating events, and its tag identifiers assume proper unique values from the tag set. Then, the script of the event handler is executed (i.e., the specified events are caused). The instance of the event handler is active from the moment it has been activated until the execution of its script has been completed.

The scope of a formal parameter in the event handler heading includes the whole event handler; whereas, the scope of a tag identifier includes the whole script. Free identifiers in some part of the event handler must be known at the the containing module body. Note that formal parameters and tag identifiers declared in an event handler are local identifiers of that handler; they are not known outside the handler and there is no way of exporting their names. The following example shows how type identifiers can be used to abstract the values of some parameters:

```
record == int, int, bool ;
E, E2: event (record, int) ;

on  E (r: record, i: int) where  i<100
   par_cause
           E (r, i+1) ; E2 (r, i-1)
end ;

on  E (i1, i2: int, b: bool, i: int) where  i>=100
   seq_cause
           E (-i1, i1-i2, not (b), 0) ;
end ;
```

In the first event handler, no computation needs to be done on the values of the first three parameters of the event from class E. They are bound to one formal parameter that can only be copied in the script. In the second handler, the values are needed for some computations, so they have been bound to separate formal parameters. Note that if record were defined in a module not containing the above handlers, and were appropriately imported to the module containing the above handlers, then only the top handler would be legal. The structural details of the type identifier, record, are not known here in this case.

An event class is denoted in an event handler either by an event class identifier, or by a formal parameter of an event type (that is bound to an event class identifier). In both cases we say that the class is denoted by a *class designator*.

4.6.1 Where Clause Predicates

The language contains special predicates that can be used in the where clause of an event handler to specify constraints on the order of choosing event collections to activate instances of that event handler. These predicates are appended to the (optional) boolean expression in the where clause. The semantics of the where clause is that in order to activate an instance of an event handler for a collection of events, the value of the boolean expression must be **true** (if it exists), and in addition, all the predicates must be satisfied for those events. Several predicates can appear in the same condition; in such case they are processed from left to right. The action of evaluating the where clause (in particular, the predicates) and selecting an event collection out of the various candidates is atomic; it can be viewed as an action which takes zero time. The meanings of the different predicates are defined below.

1. **exist**: The argument specifies the name of an event class. The predicate is satisfied if at least one event of that event class currently exists; e.g.,

   on E_1 (i, j: **int**) **where** (i>0) $\wedge$ **exist** (E_2)

2. **none**: The dual of **exist**. The predicate is satisfied if no event of the specified event class currently exists.

3. **min**: The value of the (integer) argument is the minimum over all possible collec ons of existing events that match the part of the event handler heading to the left of the **min** predicate; e.g.,

   on E_1 (i, j: **int**) **where min** (i-j)

4. **max**: The dual of **min**; e.g.,

on E_1 (i, j: int) where (i<0) ∧ max (i)
{ choose an event from E_1 with maximum negative parameter }

5. **first**: The argument specifies the name of an event class (by means of an event class identifier without a formal parameter list), or an event that is a candidate to activate an instance of the event handler (by including the formal parameters of an event descriptor from the heading). In both cases, the specified event class must be one of the event classes specified in the event descriptor list of this event handler. The selected event occurred *first* among all existing events from the denoted class that match the part of the event handler heading to the left of the first predicate; e.g.,

   on E_1 (i, j: int) where (i>j) ∧ first (E_1)
   { choose the first event from E_1 for which i>j }

   on E_1 (i, j: int) ∧ E_1 (p, q: int) where (i>j) ∧ first (E_1 (i, j: int))
   { choose the first event from E_1 for which i>j, and another one arbitrarily }

6. **last**: The dual of **first**; e.g.,

   on E_1 (i, j: int) where last (E_1) { last in first out order }

Note that one of the properties of the above predicates is that if they are evaluated for the same collection of events at two different points in time their values may change (due to the possible creation and disappearance of other events).

4.7 Modules

Most features of the module have already been described. If a module does not contain an import list, none of the identifiers declared or defined outside it can be referenced within its body. Clearly, such a module is not interesting since none of its handlers can ever be activated. The other extreme case is that of a module that imports all identifiers known out of its body. This can be achieved by using the keyword all, or by explicitly listing all identifiers. This case is not useless; it allows declarations of local identifiers within this module, and exportation of some of them if needed. All identifiers known within a module body can be exported by using the keyword all in the export list.

The scope of an identifier declared (or defined) in a module body includes every event handler declared in that body (unless that handler uses a local identifier with identical name), and all inner modules that import that identifier. If this identifier is exported from the module, its scope is determined according to the above definition as if it has been declared in the body of the containing module. If an imported identifier is *not* declared or defined in the importing module, its scope within this module is determined as if it has been declared in the body of the module; otherwise, the imported identifier is not known within the module body and instead, the locally declared identifier is known there.

4.8 Programs and System Events

ıne structure of a program is identical to that of a module body. A program can be viewed as if it is contained within an implicit module. This module contains, in addition to the program itself, implicit declarations of the system event class identifiers. The system routines that handle the system events can be considered as special event handlers defined in this implicit module. System events are the mechanism by which a program interacts with the external environment. Possible examples of several simple system event class identifiers that one can expect to find in implementations of the language are:

```
print: single_use recurrent event (int) ;
                        { print and do not reply }

printc: single_use recurrent event (int, tag, event (tag)) ;
                        { print, then cause continuation event with given tag }

read: single_use recurrent event (tag, event (int, tag)) ;
                        { cause continuation event with read data, and given tag }

tick: multi_use non_recurrent event (int) ;
                        { caused each tick of the real time clock in the system
                        with increasing consecutive numbers }
```

The system event class identifier **program_start** is considered as part of the language. The system causes one event from this class when the program is started. It is pre-declared as:

```
program_start: multi_use recurrent event ;
```

4.9 Summary

This chapter has completed the definition of the language by defining the various types of expressions in the language, the scope rules for event handlers and modules, and the where clause predicates. The where clause predicates increase the expressive power of the language as shown in chapter 6. However, they somewhat complicate the implementation of the language as discussed in chapter 7.

5. Expressive Power of EBL

The expressive power of a language is a measure that refers to the ease with which algorithms can be constructed in the language. This measure is not quantitative and is rather vague. The following two chapters attempt to give the reader a feeling about the expressive power of EBL. This chapter shows how conventional language constructs; such as: goto statement, if statement, while statement, procedures, functions, variables, assignment statement, and semaphores; can be mapped onto EBL. A translation scheme from an extended language XEBL to EBL is given. The purpose of this chapter is not to encourage the use of the presented schemes, but rather to show that all those constructs can be modeled in a straightforward way and can be used in a systematic modular manner if desired.

The representation of most of the above conventional constructs in EBL requires a longer program text in EBL than in other languages which include the constructs in their repertoire. The reason for this is twofold: First, the absence of these constructs from EBL; and second, the fact that the constructs of EBL are rather primitive. The difficulties that one would encounter while trying to express the event mechanisms of EBL in another language are probably greater. The next chapter tries to balance the picture by giving some examples that are easily expressed in EBL.

The following sections show how each of the above language constructs can be mapped onto EBL when treated separately. In other words, interaction among constructs (e.g., an if statement within a while statement) is not treated. After the presentation of these basic schemes everything is put together by a translation scheme from XEBL to EBL.

In the following sections S_i denotes an arbitrary EBL script, and B denotes a boolean expression.

5.1 Goto Statement

If L is a label in the program, then one possible way to express the following construct,

```
on <event_handler_heading>  { <tag_declaration> }$^1$
   { par_cause | seq_cause }$_1^1$
            S0 ;
            goto L ;
            S1
end ;
```

in EBL is as follows:

```
on <event_handler_heading>  { <tag_declaration> }$^1$
   { par_cause | seq_cause }$_1^1$
            S0 ;
            L ;                    { cause a goto event }
            S1
end ;

L: event ;

on  L
            . . .
end ;
```

Note that if S_1 is not empty and the event handler is a sequential handler then the scheme actually presents a *fork statement*; in order to obtain a goto statement, S_1 should be eliminated. The above method can be extended to enable passing some explicit data to the target label by adding parameters to L.

5.2 If Statement

The following construct,

```
on <event_handler_heading>  { <tag_declaration> }
   seq_cause
           S0 ;
           if B then S1 else S2 ;
           S3
end ;
```

can be expressed in EBL as follows:

```
on <event_handler_heading>  { <tag_declaration> }
   seq_cause
           S0 ;
           if (B)                { select an alternative }
end ;

if: event (bool) ;
endif: event ;

on if (b: bool) where b          { alternative 1 }
   seq_cause
           S1 ;
           endif
end ;

on if (b: bool) where not b      { alternative 2 }
   seq_cause
           S2 ;
           endif
end ;

on endif                         { continue after the if statement }
   seq_cause
           S3
end ;
```

The above scheme deals with sequential handlers; the case of parallel handlers is even easier to express. The scheme should be slightly modified if S_1, S_2, or S_3 refer to formal parameters of the containing event handler. In this case, the values denoted by the formal parameters will be passed to the two event handlers which are activated by occurrences of if, simply by adding the necessary parameters to the event class identifier if. The if statement translation scheme can be easily extended to handle a case statement. In the EBL subprogram corresponding to a case statement consisting of n alternatives all n predicates can be evaluated concurrently.

5.3 While Statement

The following construct,

```
on <event_handler_heading>  { <tag_declaration> }
   seq_cause
           S0 ;
           while B do S1 end ;
           S2
end ;
```

can be expressed in EBL as follows:

```
on <event_handler_heading>  { <tag_declaration> }
   seq_cause
           S0 ;
           while (B) ;          { activate the while statement }
end ;

while: event (bool) ;

on while (b: bool) where b      { entry to the "loop" }
   seq_cause
           S1 ;
           while (B)            { another iteration }
end ;

on while (b: bool) where not b  { continue on termination }
   seq_cause
           S2
end ;
```

As in the case of the if statement, the above scheme should be slightly modified in order to express parallel handlers or if formal parameters of the original event handler are referred to in S_1 or S_2; the same methods can be applied here.

5.4 Procedures

If pr is a procedure name, then one of the ways to express in EBL the declaration of the procedure pr,

```
procedure pr { ( <formal_parameter_list> ) }¹
          { <tag_declaration> }
  do
          S0 ;
          return
end ;
```

is as follows:

```
epr: event (event (tag), tag { , <type_list> }¹ ) ;

on epr (c: event (tag), t: tag { , <formal_parameter_list> }¹ )
          { <tag_declaration> }
  { par_cause | seq_cause }¹₁
          S0 ;
          c (t)                  { signal procedure termination }
end ;
```

The following procedure call,

```
on <event_handler_heading>   { <tag_declaration> }
  seq_cause
          S1 ;
          call pr { ( <actual_parameter> { , <actual_parameter> } ) }¹ ;
          S2
end ;
```

will now be expressed as:

```
con: event (tag) ;                { continuation event }
jpr: event (tag, ... ) ;          { for passing state to continuation point }

on <event_handler_heading>  { <tag_declaration> }
        tpr: tag ;                { declare a unique tag for this call }
   seq_cause
        S1 ;
        epr (con, tpr { , <actual_parameter> { , <actual_parameter> } }1 )
                                  { call pr } ;
        jpr (tpr, ... )           { to join the continuation point }
end ;

on con (t1: tag) ∧ jpr (t2: tag, ... ) where t1=t2 { when result arrives }
   seq_cause
        S2
end ;
```

The above scheme for mapping procedure calls allows recursive procedure calls, and concurrent activations of the same procedure. It uses continuations as the mechanism for "returning" from the procedure call. The scheme allows passing state information to the continuation point by communicating to that point an event from the class jpr. The scheme uses tag parameters for distinguishing between different calls to the procedure. A tag identifier is declared within the event handler containing the procedure call. For every instance of the event handler which invokes the procedure, the (local) tag identifier assumes a unique value; thus, no ambiguity occurs on joining the results at the continuation point. Therefore, recursive procedures and concurrent activations of procedures can be used. In procedures that do not require all the above features, the mechanism can be simplified.

5.5 Functions

In order to express a function in which the return statement has an argument specifying a value to be returned as the function result, only slight modifications are needed in the scheme for expressing procedures described above. The first parameter of the event class identifier epr will include in its type list in addition to a tag parameter also the result parameter. The event c(t) in the script of the function definition will be modified to c(t, ...) where the dots represent the returned expression. Some other trivial modifications resulting from the above changes need to be done. The scheme proposed here can be easily extended to express a *function returning several values*.

5.6 Variables

EBL has no conventional variables and no assignment statement. Despite this fact, both of those can be easily mapped onto EBL and the effect of mutable objects can be achieved. This chapter shows how variables can be modeled through the use of **single_use** events. The next chapter includes an example (the readers writers problem) showing how **multi_use** events can be used to model the effect of mutable objects. This section deals with *references to variables* and the next section with *assigning values to variables*. **single_use** events can be used to "store" the variables' values. In the following declarations of variables,

> v: <basic_type> ;
> rv: **record** (<field_list>) ;
> av: **array** [<index_range> *{* , <index_range> *}*] **of** <basic_type> ;

the syntax of <field_list> is similar to that of <formal_parameter_list>, with the only differennce that a field must be of basic type; and the syntax of <index_range> is <constant> .. <constant>. The declarations will be replaced by:

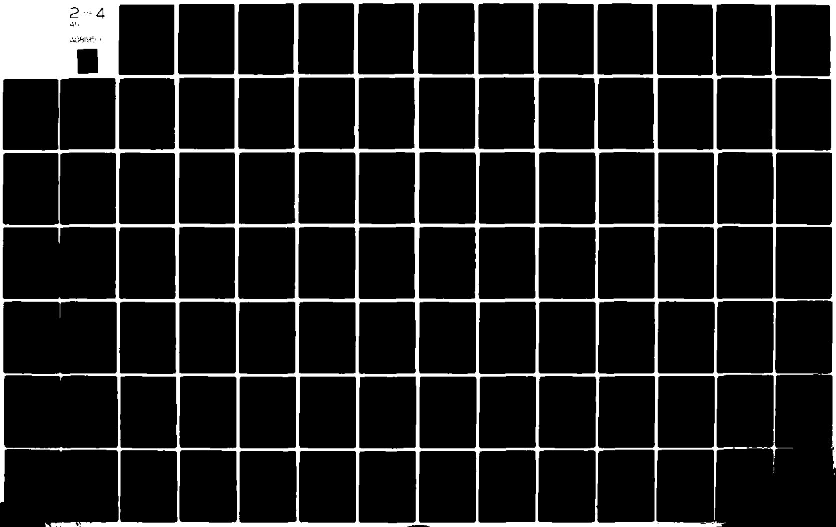
```
ev:  event ( <basic_type> ) ;
erv: event ( <basic_type> { , <basic_type> } ) ;
eav: event ( int { , int } , <basic_type> ) ;

on program_start
   { par_cause | seq_cause }$_1^1$
          ev  ( ... )              { some initial value of basic type } ;
          erv ( ... )              { some initial value for every field } ;
          { For each element of the array av, cause an event from the event
                     class eav. Cause all elements of the array, either
                     explicitly or use the while statement scheme }
end ;
```

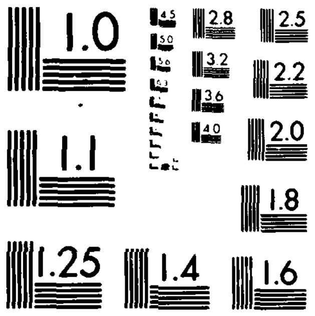
Each conventional *simple variable* (a variable of a basic type), *record variable*, or *array* is represented by an event class identifier of a **single_use** type. Exactly one event from every class is caused (one event object is created) initially for every simple variable, record variable, or array element. The parameters have some default values according to their types, or according to the values to which the variables were to be initialized. For convenience the name selected for each event class identifier is obtained by adding the character "e" to the variable's name; thus, keeping the original name for a later use, as will be shown shortly.

Now the way to reference variables is examined, starting with references to variables of a basic type in expressions. The idea is simple: if several variables appear in an expression, the modified names of those variables are included in the heading of the event handler containing the expression; thus, the current values of those variables are read (and destroyed) when the instance of the event handler is activated, and then restored when the script is executed. Assume that a, b, ... , z are the names of the variables included in an expression <exp>, then the following construct,

AD-A081 950 MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/6 9/2
THE EVENT BASED LANGUAGE AND ITS MULTIPLE PROCESSOR IMPLEMENTAT--ETC(U)
JAN 80 A REUVENI N00014-75-C-0661
UNCLASSIFIED MIT/LCS/TR-226 NL

2 ‑ 4
AD
A081950




1.0
4.5
5.0
5.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

```
on <event_descriptor_list> { where <condition> }$^1$
        { <tag_declaration> }
  { par_cause | seq_cause }$_1^1$
        S$_1$ ;
        { a statement containing <exp> } ;
        S$_2$
end ;
```

will be represented as:

```
on <event_descriptor_list> ∧
        ea (a: <basic_type> ) ∧ ... ∧ ez (z: <basic_type> )
                        { where <condition> }$^1$
        { <tag_declaration> }
  { par_cause | seq_cause }$_1^1$
        S$_1$ ;
        ea (a) ;            { restore the variable }
        .
        .
        .
        ez (z) ;            { restore the variable }
        { a statement containing <exp> } ;
        S$_2$
end ;
```

In the above scheme, the expression itself is written unchanged (as a result of the selected naming scheme). Of course, naming conflicts should be resolved, but this is a trivial problem.

In order to refer to the field fi of record variable rv in an expression, the following event descriptor,

erv (... , fi: <basic_type> , ...)

can be included in the heading. In addition, the following event,

erv (... , fi, ...) ;   { restore the record variable }

is added to the script of the event handler. In order to refer to some array element in an expression, e.g.,

av [<index>$_1$, ... , <index>$_n$]

where <index> is an integer expression, the modified names of all variables appearing in indexes of the array are simply included in the heading of the event handler; in addition, an event descriptor for the array element itself is added and the where clause is modified. The added event descriptor is.

eav (i_1, ... , i_n: int, av: <basic_type>)

and the following clause is appended to the boolean expression in the where clause:

and (i_1 = <index>$_1$ and ... and i_n = <index>$_n$)

The script will include the event:

eav (i_1, ... , i_n, av) ; { restore the array element }

Function calls within expressions in some statement S, can be moved to assignment statements before S such that each assignment statement in the (modified) program contains at most one function call, which is at the outermost level of the expression. The section on assignment statement deals with this case.

5.7 Assignment Statement

Since one event object carries the current value associated with a simple variable, the current record associated with a record variable, or the current value associated with an array element, the assignment consists of destroying the current event object (variable object) and creating a new one. Thus, the effect of mutable objects is achieved by destroying the current objects and creating updated ones. First, consider the case in which the right hand side of the assignment statement does not contain a function call. This case is further subdivided according to the left hand side of the assignment

statement which can be a simple variable, a record variable component (e.g., rv.fi), or an array element. Assignment to the simple variable v,

```
on <event_handler_heading>  { <tag_declaration> }
  { par_cause | seq_cause }
         S1 ;
         v := <exp> ;
         S2
end ;
```

where <exp> refers to variables a, b, ... , z will be represented as:

```
on <event_handler_heading>  { <tag_declaration> }
  { par_cause | seq_cause }
         S1 ;
         L                    { goto the assignment handler }
end ;

L:  event ;                   { event for activating the assignment handler }

on  L ∧ ev (v: <basic_type> ) ∧ { the assignment handler }
         ea (a: <basic_type> ) ∧ ... ∧ ez (z: <basic_type> )
  { par_cause | seq_cause }
         ea (a) ;             { restore the variable }
         .
         .
         .
         ez (z) ;             { restore the variable }
         ev ( <exp> ) ;       { assign new value to v }
         S2
end ;
```

Assignment to record variables and array elements can be expressed in a similar fashion. Parallel assignments to several fields of a record variable can be easily achieved by changing more than one parameter in the event that assigns the new record to the record variable. In fact, using this idea one can implement several conventional variables as distinct fields in a newly created record variable and then have an atomic *parallel assignment* to several variables.

As stated previously, the program can be modified such that each assignment statement contains at most one function call (which is at the outermost level of the expression on the right hand side). The cases in which the right hand side of the assignment statement is such a function call are dealt with by combining the scheme presented in the section on functions with the scheme presented above.

5.8 Semaphores

This section shows how operations on semaphore variables can be expressed in EBL. A general semaphore variable can be implemented as an event class identifier of a single_use recurrent type. The declaration of semaphore variable s,

```
s: semaphore ;
```

is replaced by:

```
es: event ;
```

The V operation,

```
V (s) ;
```

is replaced by:

```
es ;
```

The P operation in the following event handler,

```
on <event_handler_heading>  { <tag_declaration> }
   seq_cause
         S1 ;
         P (s) ;
         S2
end ;
```

is replaced as follows:

```
on <event_handler_heading>  { <tag_declaration> }
   seq_cause
         $S_1$ ;
         L                  { activate the P handler }
end ;

L: event                    { event for activating the P handler }

on L ∧ es                   { the P handler }
   seq_cause
         $S_2$
end ;
```

General semaphore variables are often used to control the allocation of identical resources (e.g., printers). The ability to attach parameters to semaphores in EBL greatly simplifies the allocation of such resources, since the waiting process can get some information on the specific allocated resource (e.g., its logical number) when it is allowed to proceed. Multiple P operations [Di-71, Pa-71] can be easily expressed in the language. The P handler above is written as:

```
on L ∧ $es_1$ ∧ ... ∧ $es_n$        { the multiple P handler }
   seq_cause
         $S_2$
end ;
```

The desired semantics is achieved since the activation of an instance of an event handler is an atomic action. An example of the use of the multiple P operation, the five dining philosophers, is given in the next chapter.

Earlier it was shown how conventional variables can be mapped onto EBL by using event class identifiers of a single_use recurrent type. In the scheme proposed there, a reference to a variable has been always replaced by an event descriptor in the event handler heading, and a restoring event in the script. These operations are actually P and V operations; therefore, in many cases no additional synchronization operations are needed.

One can think of that scheme as waiting for the variables to be accessible, using them, and freeing them when the operation is done.

5.9 Extensions to EBL

Each of the previous sections in this chapter treated separately one conventional language construct. The rest of this chapter demonstrates how a program containing many of these constructs can be translated to EBL. The following section describes several extensions to EBL, resulting in an extended language XEBL. XEBL is not proposed as an improvement to EBL, but as a concrete example for demonstrating that such a translation is feasible. The remaining sections outline an algorithm for translating a program from XEBL to EBL. The translation is done in two phases: a translation from XEBL to its kernel, and a translation from this kernel to EBL.

5.9.1 Extended EBL

This section does not intend to give a precise definition of XEBL. A general description of the language is sufficient for the following discussion. In EBL the only kind of a program unit is an event handler. In XEBL a program unit can be an event handler, a procedure, or a function. Program units cannot contain one another; however, they can be contained in modules. The structure of a program unit is similar to that of an event handler. The body of a program unit contains (optional) tag declarations and a script.

A script is a list of statements. There are simple statements and compound statements; a compound statement is a sequence of simple statements delimited by statement brackets. The possible simple statements are: cause statement (whose structure is cause <event>), goto statement, if statement, while statement, procedure call, assignment statement, P operation, and V operation. Statements can contain other statements, like in block structured languages; those which are not contained in other statements are of level zero. Like in EBL, a script can be either parallel or sequential. In the first case execution of the statements of level zero is not ordered (in this case the script contains several parallel script branches), and in the latter case it is ordered (in this case the script contains one script branch). The execution of a script branch is always sequential, even in case of a parallel script.

The target of a goto statement (the label) must be in the script containing the goto statement. A goto statement in one branch of a parallel script cannot specify as its target a label within another branch of that script. There are simple variables (variables of basic types), record variables, and array variables. Variable declarations cannot appear within program units. Variables and function calls can appear in any expression except in event handler headings.

5.9.2 Translation from XEBL to its Kernel

The main three steps in the transformation of a program from XEBL to its kernel are discussed now. First, every while statement is transformed to an equivalent statement sequence which uses an if statement and goto statements. Second, function calls in actual parameters of cause statements, procedure calls, and function calls are eliminated. This is done by adding appropriate assignment statements before the original statements, and declaring new variables as necessary. Third, every assignment statement is transformed to an equivalent sequence of assignment statements. The left hand side of each assignment statement contains no function calls (in array indexes), and the right hand side contains at most one function call (which is at the outermost level of the expression).

The transformations are done in a way which preserves the various script branches in each script. This is achieved by adding statement brackets as necessary.

5.9.3 Translation from the Kernel to EBL

The algorithm for translating a program from the Kernel of XEBL to EBL is outlined in this section.

1. Replace every variable declaration by an appropriate event class identifier declaration. Add event handlers for causing initial events from these event classes. This initialization is triggered by the initial event program_start. In case of array variables, perform the initialization by using the translation scheme for a while statement presented earlier in this chapter.
2. Transform every procedure or function declaration to an event handler. At this step only transform the headings and create a new event class identifier for each procedure. The script is handled in other steps of this algorithm.
3. Split every statement containing a function call into two statements: a function call (identical to a procedure call), and a special assignment statement whose right hand side is function_return.
4. Find the basic blocks in each script and build a flow graph [Ah-77]. A basic block starts on: (1) the beginning of each script branch; (2) each labeled statement which is the target of at least one goto statement; (3) the beginning of each arm in an if statement; (4) each P operation; (5) each statement immediately following the end of a basic block (as defined next) in a sequence of statements.

A basic block ends in the following cases: (1) after the last statement in each script branch; (2) after each goto statement; (3) after the test in each if statement; (4) after each procedure or function call; (5) after each statement immediately preceding the beginning of a basic block (as defined earlier) in a sequence of statements.

5. Attach a unique label L to the beginning of each basic block B (if needed). Add goto statements at the end of basic blocks leading to B (according to the flow graph), except in basic blocks which terminate by a goto statement, a procedure call, or a function call. In a block leading to B, which ends with a procedure call or a function call, L will be used as a continuation in the event replacing the call.

6. For each basic block B (except in the cases defined next) whose first statement is labeled by a label L, create a new event handler whose event descriptor list contains L. The exceptions are: (1) B is the first basic block in a sequential script; (2) B contains exactly one statement, a cause statement, a goto statement, or a V operation, and it corresponds to a complete script branch in a parallel script.

7. In a parallel script, for each script branch which was transformed to at least one event handler (in the previous step) replace the script branch by a goto statement. This goto statement is in fact a fork statement; its target is the event handler corresponding to the first basic block in the script branch.

8. At this stage the program is in a form which is suitable for a simple application of the translation schemes for each of the constructs as discussed earlier in this chapter. First, references to variables appearing in expressions and assignment statements are dealt with. The structure of cause statements is modified

according to the syntax of EBL (elimination of the keyword cause). Then, procedure calls, function calls, and returns from procedures and functions are treated. At this stage, goto statements and if statements can only appear at the end of a script branch. Moreover, an if statement has the restricted form:

if <bool_exp> then goto $\langle label \rangle_1$ { else goto $\langle label \rangle_2$ } ;

The translation scheme for goto statements and if statements can be applied.

The translation algorithm generates new identifiers in various steps. We assume the existence of some simple mechanism for generating unique identifiers. Note that formal parameters of a program unit in XEBL can be referenced in statements which move to new event handlers in the translation process. The referenced parameters should be passed to these event handlers by the activating events.

5.10 Summary

The language does not contain conventional constructs such as: variables, assignment statement, goto statement, iteration constructs, procedures, functions, and semaphores. The reason is that these constructs can be easily modeled in the language. A scheme for systematically translating a program containing these constructs to EBL has been developed in this chapter.

6. Classical Examples

This chapter shows how some classical problems can be solved in terms of EBL. Its purpose is twofold: first, to show the ease of expressing the solutions, and more important, to lead through each example to some interesting observations on the language.

6.1 "Recursive" Linear Fibonacci

This section demonstrates one of the possible applications of multi_use non_recurrent events, computing functions by tables, by presenting a subprogram for computing fibonacci(n). (A similar solution appears in [Ko-79].) If the event F(n) is caused, the handlers cause RF(n,s), where n is the original argument, and s is equal to fibonacci(n).

```
F, JF: multi_use non_recurrent event (int) ;
RF: multi_use non_recurrent event (int, int) ;

on F (n: int) where n=0 or n=1               { basis }
   par_cause    RF (n, n)
end ;

on F (n: int) where n>1                      { induction step }
   par_cause
            F (n-1) ;
            F (n-2) ;
            JF (n)                           { for joining the results }
end ;

{ the joining event handler }
on JF (n: int) ∧ RF (n1, s1: int) ∧ RF (n2, s2: int)
                                             where  n1=n-1 and n2=n-2
   par_cause    RF (n, s1+s2)                { add the two subresults }
end ;
```

The above handlers can be used as follows:

```
on        . . .
          . . .
          F (i) ;                  { activate (if necessary) }
          L (i)                    { for joining the result }
end ;

L: event (int) ;                   { for activating the continuation handler }

{ the continuation handler }
on L (n1: int) ∧ RF (n, s: int) where n1=n
          . . .                    { use the result }
end ;
```

The structure of the program brings to mind the known recursive implementation of the function but the semantics of EBL's events causes a unique behavior. Since **multi_use non_recurrent** events are employed, for each n at most one event of the form F(n) can exist and once such event is caused it remains forever. Thus, for each n at most one of the first two handlers is activated since their activating conditions are mutually exclusive. For each n the joining event handler is activated at most once. This can be proved by showing that for each n the following properties hold:

P1. There is at most one event of the form JF(n).

P2. There is at most one event of the form RF(n-1,s1).

P3. There is at most one event of the form RF(n-2,s2).

Property P1 holds since JF is a **multi_use non_recurrent** event class identifier. Properties P2 and P3 can be shown by proving that:

P4. For each n at most one event of the from RF(n,s) exists.

Property P4 can be proved by induction on n. One consequence of property P4 is that no two different results can be returned by the above program for the same n.

Since the joining event handler is activated at most once for each n, the normal exponential behavior of a recursive implementation of the fibonacci function is not manifested here. In fact, after RF(n,s) is caused (i.e., after the computation of fibonacci(n) has been completed) the results of fibonacci(i) for all $0 \leq i \leq n$ are remembered forever and there is no need to compute them again. Note that the "table" is empty when the program is started, and is gradually built as the computation proceeds.

The number of events caused when computing fibonacci(n) is *at most linear* in n, and in many cases is just *one event*, L(i). Clearly, we only see here a time space tradeoff; a faster response is achieved at the expense of storing the results. Each time a **non_recurrent** event is caused, the conceptual list of events from that class has to be searched in some form or another and the search time is a function of the size of the list. In order to activate the continuation event handler, the conceptual list of results (events of class RF) has to be searched. The above searches can be done in linear time, or, if the compiler is smart and implements the event lists in some efficient structure, in less than linear time; if sufficient storage is allocated the searches can be done in a constant number of operations. In summary, this example shows how computing functions by tables can be easily expressed in the language.

6.2 The Readers Writers Problem

The problem of readers and writers was suggested originally in [Co-71] and then considered in many papers dealing with synchronization mechanisms; e.g., [Ho-74, Gr-75]. We could not resist the temptation to try to solve the problem in terms of EBL, and this led to some interesting observations. The first question was whether the problem can even be stated in terms of EBL event mechanisms. The problem involves sharing a database among readers and writers, and it is not clear at all how to model it in EBL. The only scheme so far presented for creating the effect of mutable objects relies on single_use events; however, if such a solution is adopted here then there is no concurrency at all even among readers. The reason is that the only way a reader can read a single_use event is by using it; after doing so, the event disappears and the reader has to recreate the event object before another reader can access it.

Surprisingly enough, multi_use recurrent events can model the desired behavior of a database. Let us assume the database consists of an array of records; the array will be represented by an event class identifier of an appropriate multi_use recurrent type. The sharing of data among readers can be achieved if each reader always reads the "latest version" of the appropriate object; i.e., by using the predicate last of EBL. A writer modifies an item in the database simply by causing an event from that class. The subprograms for the writers and the readers for this case are shown below:

```
        db: multi_use recurrent event (int, ... ) ;  { database. The first parameter
                                    is the array index, the following are the record fields }
        read: event (int) ;                  { for activating the reader handler.
                                             The parameter specifies an array index }

WRITER:

        on          . . .
                    . . .
                    db (index, ... )         { write a record }
        end ;
```

READER:

```
        on read (ind: int) $\wedge$ db (index: int, ... ) where (ind=index) $\wedge$ last (db)
                  . . .
                  . . .                    { use the read record fields }
        end ;
```

Notice that reading by a reader has no effect on the database. Also, no special synchronization is required among writers; a writer writes whenever it wants. The action of writing a record by a writer can be viewed as atomic (as defined for example in [Re-78]) since there is a point in time before which the new record is not yet written and after which it is written. This point (corresponding to the commit point of an atomic action) is the time at which the event is caused (the corresponding event object is inserted into the conceptual event list) and thus becomes the last existing event.

Let us examine the properties of the above solution (i.e., what problem does it solve?). This is clearly a restricted version of the known versions of the problem since the database *consists of disjoint objects (array records)*. Other restrictions stem from the fact that no mutual exclusion is used. A writer can write several records but each write operation is a separate atomic action; it cannot write several records in one atomic action. Thus, the maximal consistent unit of information in this scheme is an array record; this is the reason why a reader reads only one record in each read operation in the above scheme. Another limitation is that a writer cannot read any information of the database and perform a write operation all in one atomic action.

The restrictions of the above scheme cannot be ignored. However, there are applications for which those restrictions are not applicable. More importantly, the solution has the following interesting properties: it is extremely simple, *readers and writers can* *coexist* (i.e., *maximal concurrency is obtained*), and the delay associated with a read or write operation is minimal. Note that if all references to the database in a program use the above scheme (in particular the predicate last), a clever compiler can decide to implement the database as an array, or some similar data structure, without keeping the whole history of the database (as implied by the definition of multi_use events).

The next question is: can the above approach be extended to a less restricted version of the problem? The answer is positive; consider now the case in which the maximal consistent unit of information read by a reader is an array record (as in the previous version). However, a writer can also read several records when entering the database and leave the database consistent after modifying several records; i.e., a writer can read and modify several records in an atomic action. The trick here is to create two databases: one, for the readers, identical to that of the previous version, and the second, for the writers, consisting of single_use events which carry the data and provide (only the necessary) mutual exclusion among the writers.

```
rdb: multi_use recurrent event (int, ... ) ;   { readers database. The first
          parameter is the array index, the following are the record fields }
wdb: single_use recurrent event (int, ... ) ; { writers database.
                                 The parameters are as in rdb }
read: event (int) ;              { for activating the reader handler.
                                 The parameter specifies an array index }
write: event ( ... ) ;           { for activating the writer handler specifying
                                 which records to modify }
```

WRITER:

```
on write ( ... ) ∧ wdb ( ... ) ∧ ... ∧ wdb ( ... ) where B
                    { B selects the needed records of the array wdb }
   seq_cause
        rdb ( ... ) ; . . . ; rdb ( ... ) ;      { first, modify readers database }
        wdb ( ... ) ; . . . ; wdb ( ... )        { next, update writers database }
end ;
```

READER:

```
          on read (ind: int) ∧ rdb (index: int, ... ) where (ind=index) ∧ last (rdb)
                    ...
                    ...                        { use the read record fields }
          end ;
```

The correctness of this scheme can be easily understood by observing that it basically uses the two phase lock protocol described in [Gr-78]. This solution is still simple in comparison to the known solutions to the various versions of the problem. It provides a minimal delay for readers, which can coexist with writers; and allows several writers to coexist if they access mutually exclusive records of the database.

Note that the schemes presented above can be modified in several ways. For example, in order to decrease space overhead wdb can carry only an index parameter and not the data itself; i.e., act as a semaphore array. Also note that the known solutions to the various versions of the problem can be translated to EBL; we shall not do so since it does not provide any deeper insight.

6.3 Airline Reservation System

This section contains an example of a simplified airline reservation system. The system handles concurrently several flights, each of 100 seats, and recognizes several requests: *info*, which returns the current number of available seats in flight number f; *reserve*, which reserves n seats in flight number f; *cancel*, which cancels reservation of n seats in flight number f; and *init*, the initial command specifying the number of flights in the system. Some of the requests expect an answer and this is achieved by passing a continuation event class identifier as a parameter in the requesting event.

```
module                                          { for airline reservations }
        export: info, reserve, cancel, init ;
        intp == int, int ;                      { integer pair type identifier}
        ct == event (intp, bool) ;              { continuation type identifier }
        info: event (int, event (intp)) ;
        reserve: event (intp, ct) ;
        cancel, cnt: event (intp) ;
        init: event (int) ;

        on info (f: int, c: event (int, int)) ∧ cnt (fl, l: int) where  f=fl
           par_cause
                   cnt (fl, l) ;                { do not modify the counter }
                   c (fl, l)                    { send the reply }
        end ;

        on reserve (f, n: int, c: ct) ∧ cnt (fl, l: int) where  f=fl and l>=n
           par_cause
                   cnt (fl, l-n) ;              { decrement the counter }
                   c (f, n, true)               { positive reply }
        end ;

        on reserve (f, n: int, c: ct) ∧ cnt (fl, l: int) where  f=fl and l<n
           par_cause
                   cnt (fl, l) ;                { do not modify the counter }
                   c (f, n, false)              { negative reply }
        end ;

        on cancel (f, n: int) ∧ cnt (fl, l: int) where  f=fl
           par_cause
                   cnt (fl, l+n)                { increment the counter }
        end ;

        on init (n: int) where n>0
           par_cause
                   init (n-1) ;                 { iterate }
                   cnt (n, 100)                 { 100 seats for each flight }
        end ;
end ;                                           { of airline reservation module }
```

A counter is associated with each flight; this counter is an event from the class cnt. Each of the counters also provides for mutual exclusion among concurrent requests for accessing the data associated with the corresponding flight. The system is implemented as a module in order to protect the inner event class identifiers. The structure of the module is simple, and the behavior of its event handlers is self explanatory. The system is of course simplified and not a realistic one, but it shows the kernel of a possible more realistic system.

EBL's fairness rules (as defined in chapter 2) guarantee that each request is eventually processed. This program does not attempt to control the order in which requests are processed. For example, reservations are not necessarily handled in the order they are issued. In order to get a more fair solution, the where clause of the second event handler can be changed to:

where (f=fl and l>=n) ∧ first (reserve)

In this case, reservations which can be positively answered are handled on a FIFO basis.

6.4 Disk Head Scheduler

The problem of the disk head scheduler is presented in [Ho-74]. Basically, the problem is to reduce the average waiting time of a process wishing to access the disk at some cylinder. The first suggestion considered there is to select the process wishing to move the disk heads the shortest distance. This solution is not accepted there, since a process wishing to access a cylinder at one edge can starve forever.

The solution shown there minimizes the frequency of changing the direction of the heads movement, like the behavior of an elevator algorithm. The solution is implemented by using the monitor construct, and it should not be difficult to express it in EBL. However, let us return to the original (rejected) solution and see if and how it can be expressed using each of the following mechanisms: (1) EBL event mechanism, and (2) the monitor construct.

In the following EBL subprogram, the requests are represented by events from the class *request*. Each process wishing to access the disk causes a request event specifying the destination, and a continuation event class identifier. The continuation event is caused by the scheduler when it decides to select the process. When the process decides to free the disk, it causes an event from the class *release*, which contains the current heads position.

```
request: event (int, event (int)) ; { parameters: destination and continuation }
release: event (int) ;              { the parameter is the current heads position }

on release (headpos: int) ∧ request (dest: int, c: event (int))
                                    where min (abs (headpos-dest))
   par_cause    c (dest)            { allow the process to continue }
end ;

on   program_start
   par_cause    release (0)         { initial heads position }
end ;
```

As can be seen, the whole scheduling algorithm is implemented by one trivial event handler and an additional initialization. In order to implement such an algorithm using the monitor construct, some data structure must be added for keeping the destinations of the waiting processes; in addition, a monitor procedure for searching the data structure in order to select the next request must be explicitly written, and the solution is not elegant. The scheduled waits, which are used in [Ho-74] in his solution to the original problem, do not contribute anything to the problem we consider here.

Let us analyze the reasons for the difference in the ease of expressing the algorithm in the two cases. The first, and seemingly obvious reason is the existence of the predicate min in EBL. However, the scheduled waits offer a similar mechanism: choosing the waiting process having the maximal priority (minimum value of a parameter associated with the waiting process); therefore, this reason is not so clear.

The second reason, the difference between EBL's min predicate and the scheduled waits, is the *fundamental* one. The scheduled waits mechanism allows associating a priority with the waiting process; however, this priority is evaluated when the process is suspended (when executing a wait statement) and cannot be modified dynamically. Scheduled waits can be modeled in EBL by using the min or max predicates; however, in EBL, the priority of a waiting process (a requestor in the example) is evaluated dynamically (in the where clause) and is more powerful. Now it should be clear why the scheduled waits cannot be used to solve the problem considered here.

The example in this section suffers from the possibility of starvation. Several comments can be made on this. First, the problem can be eliminated by constructing a slightly more complicated scheduler that makes use of an additional parameter of each request: a parameter obtained by reading the latest tick value. Clearly, this is yet another scheduling algorithm, but there is no other choice since the starvation possibility is inherent in the algorithm and is not caused by choosing one mechanism or another. Second, our purpose has not been to justify the algorithm but rather to use it as some concrete example in the above discussion.

6.5 The Five Dining Philosophers

The problem of the five dining philosophers was presented in [Di-71]. It involves five philosophers spending their time in infinite cycles of eating and thinking. The philosophers sit at a round table and there is one fork between every two adjacent philosophers (total of five forks). In order to eat, each philosopher needs the fork to its left and the fork to its right. The above paper shows how deadlock can arise when solving the problem by using semaphores. A possible solution in terms of EBL is:

```
F: event (int) ;                        { forks class }
P: event (int, int) ;                   { philosophers class }

on P (lf, rf: int) ∧ F (i: int) ∧ F (j: int) where lf=i and rf=j
   seq_cause
           { eat } ;
           F (lf) ;                     { free left fork }
           F (rf) ;                     { free right fork }
           { think } ;
           P (lf, rf)                   { iterate }
end ;

on program_start
   par_cause
           { create (start) the five philosophers }
           P (1, 2) ; P (2, 3) ; P (3, 4) ; P (4, 5) ; P (5, 1) ;

           { create the five forks }
           F (1) ; F (2) ; F (3) ; F (4) ; F (5)
end ;
```

The above solution is deadlock free since it basically makes use of multiple P operations. Actually, a class of philosophers, all executing the same event handler, has been created. As implied by the problem, the order in which the philosophers are selected for restarting their cycles is not defined; however, the solution can be easily modified to be more fair by changing the where clause of the first event handler to:

```
where (lf=i and rf=j) ∧ first (P)
```

In this case, if philosopher P is selected for restarting its cycle, P is the longest waiting philosopher among all philosophers which are currently waiting and both of the forks they need are not in use.

6.6 Summary

This chapter and the previous one have explicitly dealt with the expressive power of EBL. The previous chapter has shown how variables can be implemented in terms of single_use events. This chapter has shown another way to implement shared mutable objects; this time in terms of multi_use events. The power and the roles of some of EBL's predicates have been demonstrated. It seems that synchronization problems and problems involving resource allocation are especially easy to solve in EBL.

7. Virtual System Implementation

The purpose of this chapter is to investigate implementation schemes which are natural to the language. A system with virtually unlimited computational resources will be selected as a concrete example. This investigation is not useless since, as will be seen in the following chapters, its results can be easily adapted to more realistic computer systems. In some places in this chapter the existence of unlimited computational resources is not exploited since the purpose is to develop schemes which can be employed in reality (after some adaptation).

7.1 Overview

This chapter assumes a virtual system having an architecture of a fully connected processor network with unlimited computational resources (processors, processors' local storage, and link capacity). An attractive implementation on such a system associates an *event class manager ECM*, with each event class identifier in the program, and an *event handler manager EHM*, with each event handler. Each manager is assigned to a distinct processor.

An ECM is responsible for events from the corresponding event class. It maintains the event objects in an event list which may be (but not necessarily) implemented as a linked list. Whenever a new event is to be caused by some instance of an event handler the corresponding ECM receives the event object and inserts it into its event list. It then may broadcast a message to all the relevant EHM's to check whether they can now find new matching event collections.

An EHM is a cyclic process that repeatedly tries to find new matching event collections. It does it by scanning the relevant event lists, by exchanging messages with the corresponding ECM's. When it finds a matching event collection it must verify that the **single_use** events in the collection have not been used so far (i.e., they still exist) and can be used by it, all in an atomic action called the *acquisition.* Then it can activate a new instance of the event handler, by allocating a new processor for this task.

An activated instance of an event handler has to evaluate parameters and to cause events. Since there is a lot of concurrency to be exploited even within a single instance of an event handler, new processors will be allocated as needed to achieve a high degree of parallelism. When an event object is built, a message is sent to the corresponding ECM, which inserts the object into its event list. The managers and the instances of event handlers communicate with each other by exchanging messages. The messages are either requests (e.g., "insert an event object into a list"), or replies (e.g., "here is the next list element").

One of the tasks of an EHM is the acquisition of **single_use** events in a matching event collection. The problem is akin to that of locking of objects in a distributed database, and a two phase acquisition (locking) algorithm can be employed. As in many existing locking algorithms deadlocks can be prevented in our case by defining a total order on all objects to be locked. The drawback of this approach is that objects are locked sequentially. A scheme in which deadlocks are prevented by using a partial order on all object classes (event classes) is developed in this chapter. The advantage of this scheme is that objects can be locked by a requestor (an EHM) concurrently, thus the acquisition action can be completed faster.

The EHM algorithm is described in this chapter in terms of a conceptual tool: the *event space*. One event space is associated with each event handler in the program. Each of the n axes of an event space ranges over all existing events from one of the n event classes defined by the event descriptor list of the event handler. Event collections can be represented as points in the n dimensional event space. Each event space changes dynamically and the role of the corresponding EHM is to continually check whether points in the event space correspond to matching event collections.

The behavior of an EHM is determined by two orthogonal properties of the corresponding event handler. The first property is the existence of **single_use** event class identifiers in the event descriptor list. If the event descriptor list contains at least one event class identifier of a **single_use** type the event handler is a *single_use event handler*, otherwise it is a *multi_use event handler*. The second property is the existence of predicates in the where clause of the event handler. According to the above properties there are 4 different cases:

Case 1: A multi_use event handler without predicates in its where clause.

Case 2: A single_use event handler without predicates in its where clause.

Case 3: A multi_use event handler with at least one predicate in its where clause.

Case 4: A single_use event handler with at least one predicate in its where clause.

In all above cases, an EHM begins a search for matching event collections after receiving a message from an ECM (saying that a change in its event list has occurred). In cases 1-3, the EHM only checks new points in its event space whereas in case 4, in general, the whole event space has to be searched. In cases 1 and 3, once a matching event collection is found an instance of the event handler is activated by the EHM. In cases 2 and 4, however, the EHM has first to acquire the **single_use** events in the event collection (by the acquisition algorithm) and only then it activates an instance of the event handler.

Before an EHM begins a search for matching event collections it has to find the current boundaries of its event space. These boundaries consist of the current boundaries of all the relevant event lists. The action of finding the boundaries cannot be implemented as a sequence of actions reading the needed values since a set of values inconsistent with each other may be obtained (examples are given in section 7.7). The boundaries must be sampled at exactly the same point in time. An implementation of such an action in a distributed system without any centralized control is not trivial. Strategies based on locks or timestamps can be employed.

An implementation of EBL must guarantee the language fairness rules. The fundamental idea in guaranteeing the fairness rules is that each EHM should detect a case in which it could use an event in many opportunities but it failed due to contention with other EHM's. When an EHM detects that it failed it acts to reduce the likelihood that it fails in the future. It increments a counter (a *failure counter*) associated with the EHM and the ECM whose event is a reason for the failure. Each ECM knows the values of the relevant failure counters. Therefore, it can give a higher priority to requests associated with the EHM whose counter has the highest value.

In the general case, an event list can be implemented as a doubly linked list. In certain cases, however, optimizations can be made. In some cases an event list can be replaced by a counter and in others by a record variable. Sorting an event list can result in a better performance in certain cases. These optimizations and others are discussed in a section 7.12.

The rest of this chapter further develops the ideas discussed in this section.

7.2 The Virtual System

The virtual system consists of a very large number of processors; each processor is directly connected to all the other processors, and is equipped with unlimited local storage. The processors are organized by some mechanism in a free processor pool. Whenever a processor is needed for some computational task, it can be immediately obtained from the pool. When the task is complete, the processor is returned to the free processor pool.

The links connecting processors are of a very high speed (unlimited capacity), a very low propagation delay, and no overhead is associated with sending (receiving) messages through them. Thus, the cost of communication between program objects residing on different processors is zero. This property of the system encourages the allocation of a distinct processor for each task in order to achieve good time performance.

It is important to observe that the above view of the system does not restrict the applicability of the analysis in this chapter to a network of processors communicating through communication links; the results apply also to a multiprocessor system containing a shared memory. The main difference is that in the latter case, program object A residing on one processor can access the local data of program object B residing on another processor without bothering B (as long as undesired interferences do not occur) and thus better performance can be achieved, unless the shared memory becomes a bottleneck.

In this system, once an object (such as a manager, an instance of an event handler, or an event object) is assigned to a processor there is no need to move it to another processor (although it may be copied to other processors). Thus, objects can be addressed by processor number plus some additional unique identifier. In the sequel $P(O_i)$ denotes the processor on which object O_i currently resides. The EHM associated with event handler H is denoted by M(H); similarly, the ECM associated with event class identifier E is denoted by M(E).

7.3 The Event Handler Manager

The role of an EHM is to repeatedly try to find new event collections which match the event handler heading. For each such collection to acquire the **single_use** events in the collection (perform the acquisition action defined earlier), and then to activate a new instance of the event handler. Before one designs an EHM algorithm answers should be given to the following questions:

1. At which points during the course of the computation should an EHM start searching for new event collections?
2. How and whether an EHM should keep information saying for which event collections the event handler has already been activated?
3. Is it sufficient for an EHM to check whether an event collection is matching once, or can it happen that at one point in time an event collection does not match, and later it matches an event handler heading?
4. How does an EHM acquire the **single_use** events in a matching event collection?

Can deadlocks occur?

The first question is discussed below. In a system like our virtual system, in which processing power is unbounded, or if time performance is of no importance, an EHM can be implemented as a process which continually checks whether there are new matching event collections. However, since the purpose of this chapter is to introduce schemes that can be easily adapted to real world systems, we shall not accept such a scheme.

The above scheme is time consuming, and it ignores the special semantic properties of the language which allow fulfilling the EHM role much more efficiently. There is no need to search for event collections unless there is a change in at least one of the event lists associated with event class identifiers appearing in the event handler heading. Not every change should be considered. For an event class identifier appearing in the event descriptor list, only insertions of new event objects are relevant. For an event class identifier appearing as an argument of an **exist** (**none**) predicate, the search should start only if the list changes from an empty (not empty) list to a not empty (empty) list. Since the managers of a given program are fixed and known at compile time, an ECM can signal a known fixed group of EHM's whenever a relevant change occurs in its event list. This signal is broadcast by the ECM to the relevant EHM's. If the event class identifier appears in the headings of n event handlers then at most n ECM's should be notified when the event list changes.

The answer to the second question depends on the types of the event class identifiers in the event descriptor list. If there is at least one event class identifier of a **single_use** type, i.e., the event handler is a *single_use event handler*, there is no need to keep information saying for which event collections the event handler has already been activated; the reason is that the **single_use** *events in such event collections are forgotten.* If all event class identifiers are of **multi_use** types, i.e., the event handler is a *multi_use event handler*, the information is needed. A way for keeping the information is described in section 7.6.

The answer to the third question depends on the existence of predicates in the where clause. If there are predicates, an event collection that does not match the event handler heading at one point in time may match at another point in time. If there are no predicates, this cannot happen. A way for keeping information about event collections which should be revisited is described in section 7.6.

7.4 The Acquisition Algorithm

The fourth problem of the previous section, acquisition of the **single_use** events of a matching event collection, poses a difficulty. On one hand, the acquisition should be an atomic action; on the other hand, several EHM's may wish to acquire the same object. The problem arises since EHM's operate concurrently in a distributed system without any centralized control. The problem is akin to that of locking objects needed by a transaction in a distributed database system, but it is simpler due to several reasons. First, in our case locked objects (**single_use** events) are never unlocked. Second, if an EHM finds that an object it wishes to lock is already locked it aborts the attempt to lock the object and it never needs to lock this object again. Third, in our case the requestors (EHM's) are fixed, and each of which tries to lock objects from a fixed group of object classes.

The problem can be solved by a *two phase acquisition algorithm* which is reminiscent of the two phase commit protocol of [Gr-78]. In the first phase, the *booking phase*, the EHM requests the relevant ECM's to mark the single_use events as *booked*. If the first phase succeeds, i.e., all needed objects are successfully booked, the second phase in which the EHM requests the ECM's to mark the events it has previously booked as *acquired* is entered.

If in the booking phase an EHM is notified that an object it tries to book is already acquired, the EHM frees all objects it has successfully booked and aborts the algorithm. If a booked object is found, the acquisition algorithm is suspended until the object is either freed or acquired. In order to successfully acquire n objects an EHM has to send at most 2n messages, assuming each ECM keeps suspended requests. The above algorithm is susceptible to deadlock. A deadlock can involve only EHM's executing the first phase of the acquisition algorithm since in the second phase, the needed objects are already booked and therefore out of the contention.

In general, deadlocks among requestors of resources can be treated in one of the following ways [Ch-74b]:

1. *Detect* deadlocks after they occur, then cure the problem, e.g., by pre-empting resources in a way which breaks the deadlock.
2. *Avoid* deadlocks by forcing each requestor to request all the resources it needs in advance. Only requests which do not lead to deadlock are granted.
3. *Prevent* deadlocks. In this case, deadlocks cannot occur due to the underlying system algorithm.

In our case, deadlocks can be prevented in most cases if all EHM's adhere to the following booking rule:

B1. Objects are booked sequentially one at a time in an order preserving a global order (chosen by the compiler) on all event class identifiers in the program.

The only case not covered by the above rule is that of more than one event handler each containing the same event class identifier of a **single_use** type in more than one event descriptor in its event descriptor list. One way to prevent deadlocks in this uncommon case is by adding the following rules to the corresponding EHM's:

B2. Every distinct **single_use** event in the event collection is booked at most once. This takes care of an event collection containing the same event more than once.

B3. Events from an event class are booked according to some fixed order on the event objects in the event list. An example of such an order is the order of insertion of the event objects into the event list (which is not necessarily identical to the order of the events in the event list)

Another way is to replace rule B3 by:

B3'. *All events from the same event class are booked by the EHM in one request,* executed as an atomic action by the ECM.

Since such a multiple book request is processed by one processor (on which the ECM resides), its implementation poses no problems.

Similar algorithms for locking objects, which do not prevent deadlocks (in the sense defined earlier), normally use some aging mechanism (e.g., timestamps) in order to prevent a requestor from waiting forever due to retries (e.g., after pre-emptions) [Ch-74b, Ba-75, St-78b]. No such mechanism is needed in our case since there are no retries. The problem of requestors waiting forever does not occur if, for example, each ECM fulfills book requests in a FIFO order (except when requests are suspended).

One should observe that the booking algorithm prevents deadlocks by defining a total order on classes of resources and not on the resources themselves. The number of those classes is fixed and known to the compiler, whereas the number of the resources in each class is in general unbounded. EHM's executing the second phase of the acquisition algorithm cannot be involved in deadlocks thus they can acquire all needed objects concurrently. The combined acquisition algorithm is therefore deadlock free.

A drawback of the previous acquisition algorithm is that objects are booked sequentially. This restriction (defined in rule B1 earlier) is sufficient for preventing deadlocks (except for the cases covered by rules B2, and B3 or B3') but can be relaxed as shown in the rest of this section. The semantics of the language can be exploited to allow concurrency in the booking phase. In a general problem of requestors locking objects, the objects are dynamically selected and may be data dependent. In our case however, the classes of objects required by each requestor (EHM) are known in advance (at compile time). Our scheme can be applied to any case where a fixed number of requestors book objects from a fixed number of object classes; the object classes are disjoint and each can change in time. Each requestor books objects from a fixed set of object classes; one or more disjoint objects from each class. The scheme will be described in these general terms (the corresponding terms in our case appear in parentheses).

Let $G = (N_R, N_O; A)$ be a bipartite undirected graph defined as follows: Every requestor (EHM) is represented by exactly one node R_i in N_R, and every object class (event class identifier of a single_use type) is represented by exactly one node O_j in N_O. The set of arcs A contains exactly one arc connecting R_i and O_j if R_i locks objects from object class O_j. An example of such a graph is given in Figure 7.1. Since G is a bipartite graph every cycle C in G contains an even number of nodes 2m and has the form $(R_{p_0}, O_{q_0}, R_{p_1}, O_{q_1}, \dots, R_{p_{m-1}}, O_{q_{m-1}})$. Every cycle in G is a potential for a deadlock. However, since each O_j represents a class of objects, a deadlock may happen only if for every path $(R_{p_i}, O_{q_i}, R_{p_j})$ in the cycle, where j=(i+1) mod m, R_{p_i} and R_{p_j} try to lock the same object from class O_{q_i}. In such case a deadlock exists, for example, when R_{p_i} has successfully locked the object from class O_{q_i} for i=0, ... , m-1. (This is reminiscent of the possibility of deadlock in the five dining philosophers problem [Di-71].)

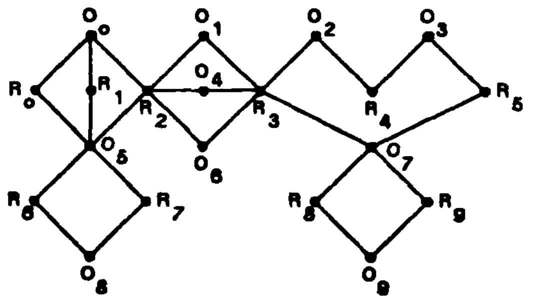


Figure 7.1 The requests graph G

In general, G is a disconnected graph consisting of several connected components. G can be separated to its n biconnected components (in the articulation points of each of its connected components); appropriate (polynomial time) algorithms can be found, e.g., in [Re-77b]. Let $S_1, S_2, \dots, S_n$ be the sets of object classes corresponding to the n biconnected components. For every i, j i≠j the set $S_i \cap S_j$ is either empty or contains exactly one element. The articulation points for G of Figure 7.1 are nodes R_2, R_3, O_5, and O_7; and the sets of object classes are $S_1=\{O_0, O_5\}$, $S_2=\{O_1, O_4, O_6\}$, $S_3=\{O_2, O_3, O_7\}$,

$S_4=\{O_5, O_8\}$, and $S_6=\{O_7, O_9\}$.

Let $G' = (N'_S, N'_O; A')$ be the reduced bipartite undirected graph of G defined as follows: Every set S_i is represented by exactly one node in N'_S. If $S_i \cap S_j = \{O_k\}$ then O_k is represented by exactly one node in N'_O; this node is connected by exactly one arc to S_i, and by exactly one arc to S_j. From the construction of G' it is clear that it is acyclic. The reduced graph for G of Figure 7.1 is depicted in Figure 7.2.

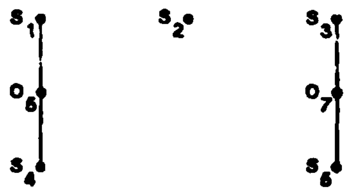


Figure 7.2 The reduced graph G'

In order to prevent deadlocks it is sufficient to define a total order on the elements (object classes) of each S_i and not on all object classes (all event class identifiers of the program). The union of these total orders defines a partial order P on the set of all object classes; the fact that the graph corresponding to P contains no cycles stems from the fact that G' is acyclic. A possible partial order for the reduced graph of Figure 7.2 is shown in Figure 7.3.

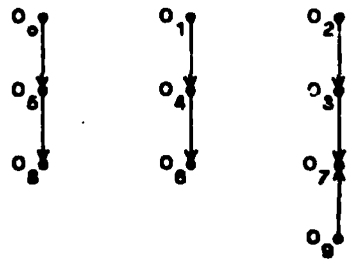


Figure 7.3 A partial order P on object classes

Rule B1 can be replaced by the following relaxed rule:

B1'. Objects are locked (booked) by a requestor (EHM) in any partial order P' consistent with P; i.e., object class O_i precedes object class O_j in P implies locking an object of class O_i precedes locking an object of class O_j in P'. Unordered locking operations can be executed concurrently.

In the relaxed locking (booking) rule, objects from distinct classes belonging to different sets S_i, S_j can be locked concurrently. In our example, requestor R_2 can lock objects from class O_0 then lock objects from class O_5, and concurrently with these operations lock sequentially objects from classes O_1, O_4, and O_6. In the relaxed scheme, the number of messages an EHM has to send in order to successfully acquire n objects is still at most 2n but the acquisition action can be completed faster due to concurrency in the algorithm.

Rule B1' specifies a sufficient condition for preventing deadlocks (together with rules B2, and B3 or B3' appropriately restated in the more general terms) but it is not a necessary condition. The reason is that nodes in N_O represent classes of objects and not objects. For example, in our case, if both H_1 and H_2 use events from two classes E_1, E_2 but H_1 specifies even parameters (in its where clause) whereas H_2 specifies odd parameters, ordering of E_1 and E_2 is not required whereas the above scheme orders them. Extending our scheme to cover such cases requires decision whether two boolean expressions (in where clauses) are mutually satisfiable. Unfortunately, the problem whether two general boolean expressions, whose terms contain relations between integer expressions (containing the operators +, -, *, and /), are mutually satisfiable is undecidable.

7.5 The Event Space

Before the various cases of the EHM algorithm are described, several terms will be introduced. Let n be the number of event class identifiers in the event descriptor list of an event handler. Each collection of events appropriately associated with the event class identifiers can be represented as a point in an n dimensional space, the *event space*, whose axes represent the n event class identifiers. Each axis ranges over all existing events from the corresponding event class (and not over all possible events from the class). One event space is associated with each event handler.

Event spaces change dynamically: An event space associated with a multi_use event handler cannot shrink in time. An event space associated with a single_use event handler H can grow and shrink in time since a part of the event space is deleted each time a single_use event of one of the event classes in the event descriptor list is used by some event handler (not necessarily by H). In the sequel, the part of an event space obtained by selecting along each axis only points between the origin and some fixed point is called a *rectangular event subspace*. The term *event subspace increment* is used when referring to the part of an event space obtained as the (set) difference of two rectangular event subspaces, one containing the other. The term *event subspace* is used in the general sense of a part of an event space. The EHM algorithms can be described and can be easily visualized in terms of event subspaces. As an example consider the following event handler heading:

on E_1 (...) ∧ E_2 (...) where ...

Figure 7.4 depicts the corresponding event space at a point in time at which 7 events from class E_1 exist and 6 events from class E_2 exist. The events are marked by x's on the axes in Figure 7.4; each of the 42 dots represents an event collection. The dashed line 1-1' (2-2') and the two axes define a rectangular event subspace containing 6 (25) event collections. The dashed lines 1-1', 2-2' and the two axes define an event subspace increment containing 19 event collections.

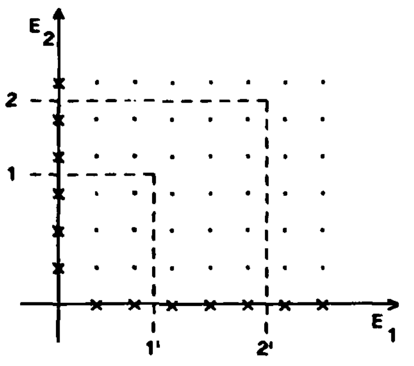


Figure 7.4 An event space

7.6 Finding Matching Event Collections

This section deals with the part of the EHM algorithm for finding new matching event collections. There are four cases according to the answers given to questions 2 and 3 of section 7.3.

7.6.1 Case 1: Multi_use Event Handler without Predicates

This case deals with a multi_use event handler without predicates in its where clause. Since there are no predicates there is no need for visiting a point in the event space more than once. Therefore, some way for covering the nonshrinking n dimensional event space is needed. Some diagonal order could be used but at each step n messages should be sent, one for each of the n ECM's, for finding the next event collection to be checked. An algorithm which always covers an event subspace increment by proceeding row by row, within each row column by column etc. in the matrix corresponding to the event space, is simpler and more efficient. At each step a message for only one ECM is needed except on the event subspace boundaries (e.g., when a new row is started).

The event space is only a conceptual tool. There is no need to actually build a matrix which corresponds to the event space (whose elements are event collections) and thus keeping the information about the event space points in memory. The needed information is easily obtained from the relevant event lists by keeping two references to the current boundaries within each event list and a reference to the currently checked event from that list.

Each time the EHM starts the search for matching event collections the previously searched rectangular event subspace is expanded along one dimension to the current boundary of the event space in that dimension. All the points added to the rectangular event subspace by this expansion (adding an event subspace increment) are checked and instances of the event handler are activated as necessary. This procedure is repeated for all n dimensions. At the end, a flag called the *restart flag* is tested to see whether messages (from the relevant ECM's) indicating that new events occurred, have been received while the EHM was performing the search. If the restart flag is set the EHM restarts the search, otherwise the search is suspended until such messages arrive.

Let us examine the number of messages M, an EHM has to send in order to cover an event subspace increment. The number of messages required on event subspace boundaries becomes less significant as the number of events in the n event lists increases; thus, these messages are ignored in the sequel. Let N_i be the number of existing events in

event list i when the previous search began, and d_i be the number of events added until the current search started. The number of EHM messages needed for covering the event subspace increment is $M = \prod_i (N_i+d_i) - \prod_i N_i$; i.e., the number of event collections in the event subspace increment. If only one event list (k) has changed then
$M = d_k \max(\prod_{i \neq k} N_i, 1)$. The max takes care of the case in which n=1.

M can also be written as $M = \prod_i N_i (\prod_i (1+\frac{d_i}{N_i})-1)$. If for all i $\frac{d_i}{N_i} << 1$, as is the case when the event lists grow (N_i increases) and the EHM responds fast enough to changes in the event lists (d_i is small) then M can be approximated by $M = \prod_i N_i \sum_i \frac{d_i}{N_i}$. If for all i N_i=N then $M = N^{n-1} \sum_i d_i$; if in addition, for all i d_i=d then $M = N^{n-1}nd$.

The efficiency of an EHM can be improved by allocating more than one processor for the event subspace search. A possible way is that the EHM allocates several slave processors for this task and assigns a disjoint part of the event subspace increment to each of them. For example, in case of a two dimensional event space each slave processor can be assigned a row in the corresponding matrix. Coordination among the slaves is required especially when predicates exist in the where clause of the relevant event handler.

7.6.2 Case 2: Single_use Event Handler without Predicates

This case deals with a single_use event handler without predicates in its where clause. The rectangular event subspace and the scheme for covering it can also be used in this case. Whenever an event collection is selected for activating an instance of an event handler its **single_use** events are forgotten; therefore, the event space does not contain any point associated with a selected (used) event collection. The covered part of the rectangular event subspace represents event collections which did not match (and therefore will never match) the event handler heading.

When a matching event collection is found its **single_use** events are acquired by the two phase acquisition algorithm described earlier in this chapter. An instance of the event handler is activated if the acquisition succeeds. When an event is acquired it cannot simply be deleted from the event list since references to it can exist in various managers. A scheme for handling this difficulty is suggested in section 7.9.

Let N'_i be the number of existing events in event list i when the previous search began. During the previous search some of these N'_i could be used (by this EHM or by others). Let N''_i ($N''_i \leq N'_i$) be the number of existing events remaining from the N'_i events when the previous search ended. Between the end of the previous search and the beginning of the current search some of the N''_i events could be used (by other EHM's). Let N_i ($N_i \leq N''_i$) be the number of existing events remaining from the N'_i events when the current search begins. For a **multi_use** event class identifier $N'_i = N''_i = N_i$; this is the reason why the distinction between these 3 values is not needed in case 1. Let d_i be the number of events added to event list i after the beginning of the previous search which are existing at the beginning of the current search.

In this case, the number of EHM messages M needed for covering the event subspace increment containing the event collections added between two consecutive searches is at most $\prod_i (N_i+d_i) - \prod_i N_i$. The reason for the inequality is that whenever a **single_use** event participating in an event collection in the event subspace increment is used, a part of the event space is deleted and it may not be needed to check some of its event collections.

In order to find out how many event collections D are deleted from the event subspace increment when a single_use event e_k from event list k is used the following algorithm can be applied.

1. If e_k is in the group of d_k added events then $D = \max\left(\prod_{i \neq k} (N_i+d_i), 1\right)$; set $d_k=d_k-1$ for later applications of this algorithm.
2. If e_k is in the group of N_k events remaining from the previous search then $D = \prod_{i \neq k} (N_i+d_i) - \prod_{i \neq k} N_i$; set $N_k=N_k-1$ for later applications of this algorithm.

The decrease in M resulting from the above event collections deletion depends on how many of these event collections have already been checked; the decrease can vary from 0 to D. The decrease can be D only if e_k is used by another EHM. If e_k is used by this EHM then the decrease can be at most D-1.

7.6.3 The Effects of Predicates

One of the effects of predicates is that an event collection that does not match the event handler heading at one point in time may match it at a later point in time. This may lead to an inefficient algorithm which repeatedly checks the same event collections. However, a deeper examination of the nature of EBL's predicates reveals that this can be avoided in many cases. This section analyzes the effects of the predicates **exist** and **none**; the results hold both for a multi_use event handler with predicates and for a single_use event handler with predicates. The effects of the other predicates are analyzed in the following two subsections.

Two trivial cases can be easily eliminated by the compiler. In the first case, the argument of an **exist** predicate also appears in the event descriptor list; in such case, the predicate can be eliminated from the where clause without changing the program behavior. In the second case, the argument of a **none** predicate also appears in the event descriptor list; in such case, the whole event handler can be deleted without changing the program behavior since no event collection will ever match its heading.

Thus, we can assume that the argument of an **exist** (**none**) predicate is an event class identifier which does not appear in the event descriptor list. This implies that the values of these predicates do not depend on the checked event collections and therefore they can be evaluated before searching of the event subspace increment is started. It is true that while the event subspace is being searched the values of these predicates may change but since the checking itself has no side effects on the values of the predicates (even when the argument is of a **single_use** type) the algorithm can proceed as if the search of the whole event subspace is done instantaneously (in zero time). Before the search of the current event subspace increment is started the **exist** (**none**) predicates are evaluated and only if they are all satisfied the search begins.

There are cases in which additional information can be obtained from evaluation of the predicates:

1. If the argument of an **exist** predicate is of a **multi_use** type and the predicate is satisfied once, it can be eliminated from the where clause since from now on it will always be satisfied.
2. If the argument of a **none** predicate is of a **multi_use** type and the predicate is not satisfied once, the EHM with all the associated information can be deleted

and the computational resources it holds can be released, since the event handler will never be activated afterwards.

7.6.4 Case 3: Multi_use Event Handler with Predicates

This case deals with a multi_use event handler with at least one predicate in its where clause. The most important observation in this section is that once a rectangular event subspace has been searched and the matching event collections have been found, there is no need to search this event subspace again. This might seem somewhat unreasonable but the following analysis of the nature of EBL's predicates shows that the observation is true. The previous section suggested a scheme for handling the predicates **exist** and **none**; this scheme does not involve revisiting points in the event space.

The next important observation is that the remaining predicates all have the flavor of minimizing (maximizing) some value. The **min (max)** predicate does it for an integer expression, and the **first (last)** predicate does it for the index of the event in the event list (assuming that an event list is ordered according to the arrival order of its objects).

If at one iteration (search) of the EHM all matching event collections of the rectangular event subspace checked at that time are found, then at later iterations no new matching points in this event subspace can be found. This follows from the optimizing nature of the predicates, from the fact that event collections selected in previous iterations are still there (since all events are of **multi_use** types), and from the fact that the event objects are immutable. This is the reason why in this case there is no need to check an event collection more than once.

Thus, each time the EHM starts a search the points in previously searched rectangular event subspaces need not be revisited. Moreover, depending on the predicates, not all new points in the event subspace increment need be checked. For example, if the where clause is **where first**(E) and a matching event collection has already been found (containing event e from class E) in an earlier search, there is no need to consider new events from class E since they must have been preceded by e. Similarly if the where clause is **where last**(E) and d events have been added to event class E since the previous search, only the latest one need be checked.

The rectangular event subspace and the scheme for covering it as described in case 1 can be used in this case with some additional information. In the most general case in which the where clause contains several instances of each predicate, we need to keep for each **first** (**last**) predicate the index of the most recently selected event (or reference to that event object if indexes are not directly available), and for each **min** (**max**) predicate, the latest minimum (maximum) value. No information should be kept for the predicates **exist** or **none**.

While examining points in the event subspace increment the arguments of **min** (**max**) predicates are evaluated and the values are compared with those kept from previous searches. Similarly, indexes can be compared for **first** (**last**) predicates. However, since in general the event lists are not implemented as contiguous arrays, the index information is not available implicitly easily. There is no need in keeping this information explicitly within the event objects. For example, if an event list is implemented as a doubly linked list references to the event objects can be kept (instead of indexes), and by comparing event object references to those kept the same results can be obtained.

In this case, the number of EHM messages M needed for covering the event subspace increment containing the event collections added between two consecutive searches is at most $\prod_i (N_i+d_i) - \prod_i N_i$ (where N_i and d_i have the same meanings as in case 1); i.e., at most the number of event collections in the event subspace increment. The inequality stems from cases (such as those described earlier) in which not all new points in the event subspace increment need be checked.

The above bound on M assumes that the EHM keeps pointers to all the points in matching event collections (having optimum values) in the event subspace increment during the search. After the search is completed instances of the event handler are activated. Another approach, in which the EHM only keeps the optimum values, requires a second search of the event subspace increment. In the second search, event collections corresponding to the optimum values found in the first search are looked for.

7.6.5 Case 4: Single_use Event Handler with Predicates

This case deals with a single_use event handler with at least one predicate in its where clause. The main difference between this case and case 3 is that here, once an event collection is selected and its single_use events are used, a part of the event space is deleted and event collections that did not match the event handler heading in the past may match the heading now. (For example, an event may satisfy a min predicate after an event with a smaller parameter value is used.) It follows that in the general case points in the event space must be revisited.

The concept of the rectangular event subspace can be used here also. Each time the EHM starts an iteration, the boundaries of the current event space are found (only points in this rectangular event subspace are checked in this iteration), and the exist and none predicates are evaluated as in case 3; only if they are all satisfied the search begins. During the search event collections are checked as in case 3. One difference, however, is that here the whole rectangular event subspace is searched and not only the event subspace increment, in contrast to case 3. Once a matching event collection is found it is treated as in case 2. After an instance of the event handler is activated and a part of the event subspace is deleted, the search of the remaining part of the (previously found) rectangular event subspace should restart in the general case.

Let N_i be the number of existing events in event list i when the current search begins. The number of EHM messages M needed for searching the rectangular event subspace until a matching event collection is found is at most $\prod_i N_i$. One reason for the inequality is that in many cases an optimum may be found without searching the whole rectangular event subspace. Another reason is the deletion of parts of the rectangular event subspace if relevant single_use events are used by other EHM's. In order to find out how many event collections D are deleted from the rectangular event subspace when a single_use event e_k from event list k is used, the following algorithm can be applied:

$$D = \max\left(\prod_{i \neq k} N_i,\ 1\right);\ \text{set } N_k = N_k - 1 \text{ for later applications of this algorithm.}$$

If e_k is used by another EHM then M is decreased by at most D.

In spite of the above, there are important cases in which there is no need to restart searching the rectangular event subspace and the search may simply proceed. First, observe that the predicates exist and none have nothing to do with restarting the search; their treatment was analyzed earlier. If there are no min or max predicates, i.e., if the where clause contains any combination of the predicates: first, last, exist, and none the order of the search in each event list is determined from the first (last) predicates. The order is forward (backward) for each event class identifier which appears as an argument of the predicate first (last).

If the argument of each min (max) predicate is a function of the formal parameters of one event descriptor, an index for the event list can be created which orders the event objects according to the function's value. The index can be maintained by the ECM which manages the event list. In this case, the order of searching the event list is determined from the index. In general, ordering the event list itself according to the function is not possible since different event handlers may have different functions for the same event class. The tradeoffs involved in sorting an event list are discussed in section 7.12.1.

The use of an index can be generalized to a case where the argument of a min (max) predicate involves formal parameters of $k \geq 1$ event descriptors. A combined index, whose elements point to event collections and not simply to event objects (each element contains k references), can be created. The combined index can be maintained by one of the k relevant ECM's, or by the EHM. If each event list contains n event objects, the size of such an index is proportional to $k*n^k$. In addition to this space overhead there is a time overhead associated with the maintenance of an index. It seems that in general, the time saving obtained by using a combined index is not justified, except possibly for very small values of k, or when time performance is of prime importance. Update of an index can be done when the event lists change or each time the EHM begins the searches.

In the above important cases, the search can proceed after an event collection is found and used. Moreover, after the search of the current rectangular event subspace is completed and all possible matching event collections are found and used, there is no need to check again points in this rectangular event subspace in future iterations of the EHM. The above observation means that in each iteration of the EHM only the event subspace increment needs to be covered and not the whole rectangular event subspace as in the general case. The number of EHM messages M needed for covering the event subspace increment containing the event collections added between two consecutive iterations is determined as in case 2.

7.7 Current Event Space Boundaries

An EHM has to find the current boundaries of the event space before it begins the search. In cases 1-3 (and in case 4 when it can be handled by searching event subspace increments) the current boundaries together with the boundaries found in the previous search define the event subspace increment to be currently searched. In case 4 (in general), the current boundaries define the the rectangular event subspace to be currently searched.

So far we have intentionally ignored the question: how are the current boundaries of the event space found? The question is not relevant to an event handler without predicates (cases 1 and 2) in which case the where clause defines properties of the event collection which are independent of any other event. Since these properties do not change in time, the only requirement from the EHM in those cases is to correctly cover the event subspace increments it finds. For an event handler with predicates (cases 3 and 4), the

EHM should compare properties of, possibly, all events in the currently checked event subspace and accordingly decide in which order to activate instances of the event handler. These properties change in time; therefore, an EHM which performs the search in nonzero time (as unfortunately all EHM's do) may select incorrect event collections unless several precautions are taken. The first precaution (which is insufficient) is to find all boundaries at exactly the same point in time.

The action of finding the boundaries cannot be implemented simply as a sequence of actions reading the needed values since a set of values inconsistent with each other may be obtained. As an example consider the following event handler heading:

on E_1 (i: int) $\wedge$ E_2 (j: int) where min (i+j)

Imagine that at some point in time three events from the classes E_1, E_2 exist: $E_1(20)$, $E_1(10)$, and $E_2(5)$. The above events have been caused after the previous EHM search has ended, and they are ordered as follows: $E_1(20)$ --c-> $E_1(10)$, and $E_1(10)$ --c-> $E_2(5)$. A naive sequential algorithm, which first finds the boundary of the event list associated with E_1 and then finds the boundary of the event list associated with E_2, may return as boundaries references to $E_1(20)$ and $E_2(5)$. These boundaries are inconsistent with the causality (or the precedes) relation because $E_1(10)$ would not be included. If event collections containing $E_2(5)$ are included in the search then event collections containing $E_1(10)$ must also be included in the search since $E_1(10)$ --p-> $E_2(5)$. An instance of the event handler would be activated for the event collection $\{E_1(20), E_2(5)\}$ instead of for $\{E_1(10), E_2(5)\}$.

Is it sufficient to sample all boundaries at the same point in time? The following example gives a negative reply by demonstrating that the predicates exist or none (together with other predicates) may cause certain anomalies if not handled correctly. Consider the event handler heading:

$$\textbf{on } E_1 \text{ (i: int)} \wedge E_2 \text{ (j: int)} \textbf{ where } \min(i+j) \wedge \text{exist}(E_3)$$

Imagine that at some point in time four events of classes E_1, E_2, and E_3 exist, ordered as follows: $E_1(10)$ --p-> $E_2(20)$, $E_2(20)$ --p-> $E_2(10)$, and $E_2(10)$ --p-> $E_3(5)$; these events have been caused after the previous EHM search has ended. An improved EHM algorithm may sample the boundaries of the lists associated with E_1 and E_2 at exactly the same point in time and return as boundaries references to $E_1(10)$ and $E_2(20)$ which are consistent with the precedes relation. However, the EHM may then naively proceed and check the value of exist(E_3) and get the result true. The boundaries together with the value of the exist predicate are inconsistent with the precedes relation. If the value of exist(E_3) is true then event collections containing $E_2(10)$ must also be included in the search since $E_2(10)$ --p-> $E_3(5)$. An instance of the event handler would be activated for the event collection $\{E_1(10), E_2(20)\}$ instead of for $\{E_1(10), E_2(10)\}$.

The problem is therefore that of implementing an action sampling the event space boundaries and the values of the exist and none predicates at exactly the same point in time. The difficulty of course stems from the fact that the system is distributed without any centralized control. Several strategies can be used to correctly implement this action.

(1) **Locks**

An obvious strategy is based on locks: the EHM locks (by shared locks) the needed ECM's (more precisely their event lists) and then reads all the needed values (sequentially or concurrently). The locking can be implemented analogously to our acquisition algorithm. Deadlocks among EHM's trying to read event subspace boundaries cannot happen since they use shared locks only. However, deadlocks involving also EHM's executing the acquisition algorithm can happen. More generally, deadlocks involving requestors which only try to lock object classes for read (by shared locks) cannot occur. However, deadlocks involving also requestors which try to lock objects for update (by write locks) can occur. For example, if requestor R_1 tries to lock (for write) objects from object classes O_1 and O_2 and requestor R_2 tries to lock (for read) object classes O_1 and O_2 a deadlock can occur.

Such deadlocks can be prevented by slightly modifying the relaxed acquisition algorithm presented in section 7.4. The graph G has now the form $G = (N_R, N_O; A, A_L)$ where N_R N_O and A are defined as in section 7.4. The added set of arcs A_L contains exactly one arc connecting R_i and O_j if all the following conditions are satisfied:

A1. R_i locks object class O_j for reading some value associated with that object class (a list boundary or a predicate value in our case).

A2. No arc connecting R_i and O_j exists in A.

A3. A Contains at least one arc which is incident on O_j.

If object class O_j is only locked for read (by shared locks) it cannot cause a deadlock and therefore there is no reason to connect it with a requestor R_i by an arc; this is the justification for condition A3. The sets of object classes $S_1, \dots, S_n$ are found as in section 7.4; a total order is defined on the elements (object classes) of each S_i and the union of these total orders defines a partial order P on the set of all object classes. The locking rule B1' of section 7.4 can be used here.

The disadvantage of the locking approach is a reduced throughput since while the locks are held, the locked event lists cannot be modified; in particular, new event objects cannot be inserted into the locked event lists. (Throughput can be defined as the total rate of event occurrences; although, a different definition is used in appendix B.)

(2) **No-Change Detection**

In this strategy, the EHM iterates at least twice through a set of actions. In each iteration, the EHM reads all the needed values (sequentially or concurrently) by communicating with the ECM's and keeps the read values. It iterates until each ECM indicates that there have been no relevant changes (to be defined shortly) in its event list between the *current reading* and the *immediately preceding reading* by the same EHM.

An ECM can keep track of such changes by allocating one bit, the *change bit*, for each interested EHM; the interested EHM's are known to the compiler. The change bit is reset each time the ECM answers a "read boundary" or "exist?" request from the corresponding EHM. The change bit is set whenever a relevant change in the event list occurs. If the event class identifier appears in the event descriptor list, then a relevant change occurs whenever an element is inserted into the event list; deletion of an element need not set the change bit, since the EHM is interested only in existing events. If the event class identifier appears in an argument of an **exist** (**none**) predicate then a relevant change occurs whenever the list becomes empty (not empty).

The main advantage of this strategy is the simplicity of its implementation. The disadvantage however, is that the EHM may iterate forever (e.g., if events of the relevant classes are caused at a rate which is higher than the EHM's iterations rate).

(3) **Timestamps**

In this strategy, an implicit timestamp parameter is added to each event object. The timestamp is added by the ECM when the event arrives. The local (logical) clocks of the various processors can be synchronized by the method described in [La-78]. In fact, for solving our problem the required synchronization is relaxed in comparison to the one required in the general case described in [La-78]. The only requirements for our problem are:

1. The timestamps reflect the precedes relation (and therefore the causality relation) between events.
2. The timestamps reflect the occurrence order implied by the predicates **exist** and **none**.

In our problem timestamps are simply a mechanism for numbering of events by monotonically increasing numbers according to the two requirements above. Thus, local clocks need not be updated each time a message tagged by a timestamp greater than the local clock arrives, but only when such a message is relevant to at least one of the two requirements above. This also implies that not every message needs to carry a timestamp.

In this strategy, the EHM M(H) requests from the relevant ECM's $M(E_i)$ (sequentially or concurrently) the needed values. All such requests are tagged by the same timestamp t_c, the current value of the local clock of P(M(H)). The purpose is to sample the needed values as of time t_c. However, due to network delays and clocks diversity (i.e., lack of perfect clock synchronization) the requests may arrive to $M(E_i)$ at local time t_i which is greater than or smaller than t_c. If $t_c \geq t_i$ (clocks diversity dominates network delays), $P(M(E_i))$ simply advances its local clock to be at least t_c, then returns the *current* requested value. If however $t_c<t_l$ (either network delays dominate clocks diversity or vice versa), $M(E_i)$ is requested to return some value which existed at some *past* point in time. The problem of maintaining history of values is addressed in [Re-78] and some of the ideas there can be used for solving our problem which is more specific.

Let us first analyze the current situation ($t_c<t_l$) for a more general case (not restricting ourselves to our specific requests). There are now two possibilities: either $M(E_i)$ has the needed value, in which case it simply returns it; or the value is no longer available, in which case a "forgotten" reply is returned. The replies are tagged by the current clock value of $P(M(E_i))$; therefore, in the latter case, P(M(H)) can increment its clock and retry. The possibility of retries is not encouraging since this strategy then suffers from the same disadvantage of strategy (2) (which has smaller overhead associated with it).

The likelihood of the need for retries can be decreased if $M(E_i)$ remembers the parts of its history which would otherwise be forgotten for at least t_m physical time units, where t_m is the sum of the maximum network delay and the maximum clocks diversity (or some estimate of this sum). Another (partial) solution is to try to avoid the problem: M(H) can first advance its local logical clock by some value Δt which is big enough, and only then follow the previous procedure. Unfortunately, when several EHM's follow this strategy a situation in which $t_c<t_l$ may quickly arise; thus, it seems that the problem has not been solved. However, it is important to observe that in this case, logical time progresses faster than physical time; the computation does not progress faster (in physical time) just because Δt increases. Therefore, the likelihood that the needed value is no longer available decreases as Δt increases.

The previous discussion was general, and at this point the specific requests in our implementation scheme and their implications are analyzed. In order to answer an "exist?" request, the ECM can keep a list of local logical times at which the state of its event list changes from empty to not empty and vice versa. Old entries in the list can be deleted (after t_m physical time units, as discussed previously), and in addition, the list can be limited to contain at most n entries (the latest change points) in order to bound its storage overhead.

In order to answer a "read boundary" request (with timestamp t_c), the ECM simply returns a reference to the latest existing event object whose timestamp t_e is not greater than t_c. There is no need to keep any information in addition to the event list in order to treat the request. Even if an event object whose timestamp t_e' satisfying $t_e<t_e'<t_c$ existed in the past (and not currently), there is no need to remember it since an EHM is interested only in existing event objects.

The behavior of the suggested timestamp based scheme is equivalent to that obtained by simultaneously sampling all needed values at a state (of the relevant event lists) which could exist at time t_c. Due to the fact that the clocks are not perfectly synchronized, this state may have not existed in reality but it could. The user has no way of checking whether this state indeed existed and this scheme exploits this fact.

The main disadvantage of this approach is a higher storage overhead; the disadvantages of the first two strategies are cured here. By selecting big enough values for the parameters t_m, n, Δt the probability of a need to retry can be decreased below any positive desired value. Moreover, attaching timestamps to event objects supports the following functions in boolean expressions (both in the where clause and in the script of event handlers).

1. A precedes function which indicates whether one event from the event collection precedes another one from the same event collection; e.g.,

   on E_1 (i: int) $\wedge$ E_2 (j: int) where i>j and E_1 (i) precedes E_2 (j)

2. A time function which returns the occurrence time of an event. Such a function can be used not only for specifying an order between two events but also for specifying by how much time one event precedes the other; e.g.,

   on E_1 (i: int) $\wedge$ E_2 (j: int) where i=j and (time (E_2 (j)) - time (E_1 (i)) > 10)

If the above functions are added to the language, the relaxed requirements for clock synchronization are not sufficient since various anomalies can happen. Assume for example that E_1, E_2 are system event class identifiers whose events occur when buttons B_1, B_2 are pressed respectively. Assume that each event from the above classes activates an instance of a corresponding event handler which causes printing the event occurrence time. Suppose clocks are not sufficiently synchronized; then if B_1 is pressed and several seconds afterwards B_2 is pressed, it may happen that the time printed for B_2 is smaller than that corresponding to B_1. Local clocks must be better synchronized to decrease the liklihood of such anomalies. The general scheme of [La-78] can be used; other schemes are described in [Re-78].

7.8 The Event Class Manager

An ECM maintains an event list which contains event objects from the corresponding event class. It is a process which handles requests from EHM's (e.g., "give next event object"), and from instances of event handlers (e.g., "insert an event object into the event list"). Each time an event object is inserted into the event list, the ECM broadcasts a message to the relevant EHM's, thus suggesting that they look for new matching event collections. Each time the event list becomes empty (not empty) it notifies the EHM's associated with event handlers which include the corresponding event class identifier as an argument of a none (exist) predicate.

7.9 Event List Organization

An event list is dynamically changing. Elements can be inserted into it, and in the case of single_use events elements can also be deleted from it. We shall concentrate on the latter case since the former is simpler. The fact that the memory associated with each processor is unbounded allows us not to do garbage collection. However, we shall not exploit this since we would like to map this implementation scheme to more constrained systems. A specific implementation of an event list will be described in order to make the following discussions more concrete.

An event list is implemented as a doubly linked list with a list head [Kn-75]. The elements in the event list are ordered according to their arrival order to the ECM. In order to support the various list operations needed to process the requests from an ECM, a list element will be in one of the following states:

ready: The normal state, the element is ready for booking.

booked: The element is booked by an EHM.

acquired: The element is acquired by an EHM; it is logically out of the list but physically still in the list (i.e., it is linked to its neighbors).

deleted: The element is both logically and physically out of the list.

The state of an element changes according to the diagram of Figure 7.5. The first three states support the two phase acquisition algorithm described earlier. The distinction between the states acquired and deleted is needed since several references to an acquired element may exist (in various EHM's searching matching event collections). Deleting the element from the list and returning it to the free storage may cause unpredictable effects. Deleting the element from the list and suspending its return to free storage until it is no longer pointed to is not sufficient since it does not allow simple tracing of the elements of a list until a particular element (which can be in the acquired state) is reached. This operation is required in our implementation scheme when searching the event subspace; the particular element is for example a boundary element.

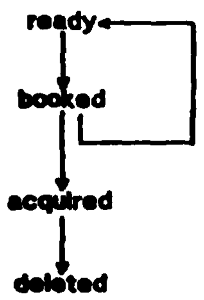


Figure 7.5 State diagram of a list element

In order to support the transfer from the acquired state to the deleted state a reference count mechanism is used [Kn-75]. Each list element contains a reference count field which counts the number of references pointing to it from EHM's. The reference count is updated when requests from EHM's are processed. It should be noted that the reference count mechanism is used to control the deletion of elements from a list and not for garbage collection purposes as is normally done.

A list element can be returned to free storage when the following two conditions are satisfied:

1. It is in the deleted state.
2. The data it carries is no longer needed; i.e., the instance of the event handler which uses it obtained all the data it requires.

Since the above conditions can be satisfied in any order, an additional bit *not-needed* is required; this bit specifies whether a message saying that the element's contents is no longer needed (and therefore can be returned to free storage after it enters the deleted state) has been received. Manipulation of the above conditions causes no synchronization problems since it is performed by one processor (on which the ECM resides). If the event list is maintained in a shared memory (directly accessible to several processors), existence of an appropriate test and set nondivisible instruction is sufficient for correct manipulation of the above conditions.

The fact that reference counts are used not for garbage collection causes some slight variations from the classical approach. The reference count mechanism should detect a situation in which an acquired list element can be deleted as early as possible in order to decrease the overhead while scanning the list. For this reason, only references from EHM's are counted; references to a list element from an instance of an event handler are not counted. The structure of an instance of an event handler is known to the compiler, which knows in particular how many times each event object is referenced in the script.

Therefore, an explicit message indicating that the event object is no longer needed can be sent by the instance of the event handler (regardless of the reference count value).

An alternative approach in which all references to a list element are counted is semantically cleaner, however, it causes a redundant overhead as was explained earlier; that is the reason why the more standard approach was slightly modified here. In both approaches, the problem of objects which are not reclaimed due to cycles cannot occur since only references from objects which are not handled by the reference count mechanism are counted. One should note that nothing prevents an EHM from copying a list element reference. Such a copying is preceded by an explicit request to increment the reference count when needed; e.g., when a reference to the boundary of the previous rectangular event space is copied to the pointer to the current list element. In some cases the explicit incrementing is not needed; e.g., when a list element reference is passed to an instance of an event handler. The decision whether to explicitly increment the reference count is made by the compiler.

7.10 Requests from an ECM

The ECM keeps requests from various sources in a request queue and processes them one at a time (although handling a request may be suspended). This section describes the main requests that a typical ECM may be required to handle. The possible requests issued by an instance of an event handler are:

Insert: Inserts an event object into the the event list; at its end.

not-needed: Indicates that the list element is no longer needed by the instance of the event handler and its *not-needed* bit can be set.

read: Reads all or some of the parameters of a list element. This request can also be issued by an EHM.

The possible requests issued by an EHM are:

book: Books an event list element. The execution of this request may be suspended if the list element is currently booked.

cancel: Cancels a previous booking of an event list element.

acquire: Acquires an event list element which was previously booked.

next: Returns the next element (satisfying some condition) in an event list. The execution of this request may be suspended if the list element to be returned is currently booked.

Two of the above requests deserve further explanation. The meaning of the insert request is more complicated in the case of non_recurrent events. An ECM should verify that the event list does not contain an object identical to the object included in the request before inserting the object into the list. Some optimization can be used here as discussed in section 7.12.1. If the event list is kept in shared memory, it is possible that the instance of the event handler trying to cause the event will search the event list itself; thus decreasing the load on the ECM. If an identical element is found, it aborts the trial; otherwise, it sends the insert request, indicating up to which point it has searched the list. At that point the ECM, which has exclusive write access to its event list, scans the rest of the event list (new elements may have been added to the list since the instance of the event handler performed its search) and treats the given element appropriately.

The *next* request is a complex request intended to support event subspace search by an EHM. The ECM searches an event list within two limits while performing some checks on the list elements. These checks could also be performed by the EHM but the communication overhead in such a solution may be high, especially for EHM's which are interested only in a small percentage of the elements of a list. The ECM can perform at least an initial filtering of the event list and thus decrease the number of messages exchanged with the EHM. The search starts at the element following an element specified in the request and proceeds (forwards or backwards as specified in the request) until an element satisfying some condition is found or the limit is reached.

The structure of the condition is limited; it is not a general boolean expression. For example, the condition can be a simple boolean relation whose terms may refer to words of a list element by denoting an offset from the beginning of the element, or to constants; e.g., according to the following syntax:

```
<condition> ::= <arg> <oper> <arg> <relational_operator> 0
<arg> ::= <constant> | <offset>
<oper> ::= add | sub | and | xor | ...
<relational_operator> ::= < | <= | = | <> | >= | >
```

Elements in the acquired state are not returned as results; they are skipped and may be deleted, depending on their reference count. The request can specify that during the search reference counts should be modified by a number specified in the request; the reference counts are updated as references move from one list element to another during the search. A special check can be performed to detect cases in which acquired elements are pointed to only by references given in the request itself. If such elements are found they are deleted from the list. If an element pointed to by one of the limit references is deleted the limit reference retreats; i.e., a new limit reference pointing to a previous element not in the acquired state is returned as one of the request results.

An interesting question is: What should an ECM do if during preforming the *next* request a booked element is encountered? First, booked elements which do not satisfy the search condition are skipped. The more important question is: how to treat booked elements satisfying the condition? Several strategies can be employed. In the first, processing of the request is suspended until the state of the element changes. In the second, such elements are returned as results. The first strategy assumes that the probability that the EHM which has booked the element will cancel the booking is low, and therefore does not return the element to the requesting EHM in order not to cause it to do fruitless work. The second strategy is based on the opposite assumption, and it therefore returns even booked elements as results. Both strategies are correct, but they may yield different performance; some experimentation is required in order to choose among them.

During processing of the *next* request, the list head is specially treated and serves as an additional limit in each direction of the search; its reference count is not modified. The first (last) element of a list can be accessed by issuing a *next* request with the list head as the current element in the forward (backward) direction. This request can be also used for finding whether the event list is empty. The first element of the list is searched for, specifying that no side effects are to occur (in particular, no reference count changes). In this case, booked elements do not cause suspending of the request, since a booked element still exists.

7.11 Fairness

So far fairness issues have been intentionally ignored. An implementation of EBL must guarantee several fairness rules (F0-F3 of chapter 2). The first question is whether the implementation scheme outlined in this chapter is fair or not. Unfortunately, our implementation scheme only guarantees fairness rule F0; each of the other rules may be violated. The principal source of the problem is contention among EHM's trying to acquire single_use events from the same event class. Suppose EHM M_i needs one event from each of the two event classes E_i and E in order to activate an instance of the corresponding event handler for i=1, 2; assume E has a single_use type associated with it. Theoretically, M_2 may never succeed in the acquisition of an event from event class E (if M_1 always wins) and thus may never use events *from* event class E_2 (regardless of the type of E_2). Depending on the rest of the program, this situation can violate F1 and F2.

For example, a P operation (as described in chapter 5) may wait forever while infinitely many P operations on the same semaphore variable successfully terminate. Similarly, a *reserve* request in the airline reservation system of chapter 6 may not be processed forever. Such situations are normally called *starvation* in process based models.

Fairness rule F3 is guaranteed in our implementation scheme for multi_use event handlers without predicates. The rule is not guaranteed, however, for multi_use event handlers with predicates. The problem arises when an exist (none) predicate contains as an argument a single_use event class identifier. The predicate may be satisfied in infinitely many periods of the clock (which is used in the fairness rules definitions), each time for a short duration. An EHM may be too slow to begin the search while the predicate is satisfied;

it therefore may never use event collections thus violating F3.

The problem can be easily eliminated by associating timestamps with events (as suggested for finding event space boundaries). An EHM can examine the value of the predicate at some past point and if it is satisfied perform the search on the event space which existed at that time. The same problem may arise in case of single_use event handlers; the above scheme can be applied here. The following scheme for guaranteeing rules F1-F2 can solve this problem too (for both types of event handlers).

The fundamental idea is that each EHM M(H) should detect a case in which it could use an event e in many opportunities (possibly together with some other events) but it failed to do so due to some race condition. If H does not contain predicates then the only reason for a failure of M(H) is that events from certain **single_use** event classes are used by other EHM's. If H contains predicates then an additional reason is that predicates may be satisfied for short periods of time which are insufficient for M(H) to select and acquire the needed events. When M(H) detects that it failed in a race it acts to increase the likelihood that it wins a similar race in the future. It basically notifies the relevant ECM's so that they can prefer it in the future.

In order to allow EHM's to detect cases in which they lose, the behavior of ECM's must be modified. An ECM should not delete an event from the event list immediately when possible, but rather, delay the deletion for a period of time T, which is long enough to enable EHM's to detect that they could use the event. T can be dynamically changed according to the sizes of the event spaces associated with the relevant EHM's. When an ECM processes the *next* request it does not skip over acquired event objects and does not suspend the processing when a booked event object is encountered. When an acquired event is thus returned it is marked as *acquired*; the EHM checks if it could use such an event but it does not try to acquire it.

What should M(H) do when it detects that it failed? Suppose M(H) finds out that event class E is one of the reasons to the failure. M(H) can then increment a counter (a *failure counter*) C_{HE} associated with the pair of managers M(H) M(E) and let M(E) know the new value. M(E) knows the values of all relevant counters and can give a higher priority to requests associated with EHM M(H) whose failure counter has the highest value. This higher priority is not sufficient since requests from M(H) may arrive too late to be served. After M(E) notifies EHM's about a change in its event list it restricts further changes in the list for a period of time T'. During this period, only changes resulting from requests made by M(H) are allowed. After this period, changes are ordered according to failure counters. T' can be dynamically changed as T above.

Let us give some details of a scheme for dynamically evaluating T' and manipulating the failure counters. Other schemes, perhaps more efficient, are possible and we describe this one only as a concrete example. In this scheme if M(H) finds during a search of its event subspace that it failed due to M(E) it increments C_{HE} by 1. Failure counters are not decremented (although other strategies are possible).

Suppose a single_use event class identifier E appears in the headings of event handlers H_1, ... , H_n. If a new event is added to the event list associated with M(E) at time t_0, M(E) sends messages to M(H_1), ... , M(H_n) and waits T' time units as described earlier. During the T' time units all above managers should be able to perform searches of their event spaces (other schemes are possible). After that interval M(E) selects M(H) whose counter has the maximum value among all managers that have requested to book events of M(E).

T' can be found as follows. The message sent by M(E) has first to propagate to all $M(H_i)$ above. A bound on this propagation time is Δ_1, the maximum network delay, which we assume is known. Each $M(H_i)$ then finds the boundaries of its event space as of a time no later than $t_0+\Delta_2$, and then searches the event subspace. Δ_2 is a bound which can be derived from Δ_1 and exact parameters of the algorithm for finding event space boundaries.

Each time there is a change in one of the event lists associated with an event class identifier E_j (which appears in the heading of H_i) the new size (number of event objects) S_j of that list is sent to M(E). Due to network delay, at time t_0 M(E) knows each S_j as of a point in time t_1, where $t_0-t_1 \leq \Delta_1$. The actual size of the event list associated with E_j that each EHM may have to scan can increase in the interval $\Delta_1+\Delta_2$ by at most $\Delta S_j = (\Delta_1+\Delta_2)R_j$. R_j is the maximum rate in which events can be added to the event list associated with E_j. It can be found from the maximum processor speed which we assume is known. $M(H_i)$ may have to scan up to $S_j+\Delta S_j$ events from the event list associated with E_j. The time required by $M(H_i)$ to complete the search is bounded by $T_i = K_i \prod_j (S_j+\Delta S_j)$ where j ranges over event class identifiers appearing in the event descriptor list of H_i. K_i depends on the heading of H_i, and on the minimum processor speed which we assume is known. Since T' should allow all managers $M(H_i)$ to complete their searches, $T' > \Delta_1+\max_i (T_i)$ is appropriate. (The term Δ_1 in the above inequality represents the propagation time of the original message sent by M(E).)

Smaller values can be selected for T' in many cases. For example, if formal parameters associated with event class identifier E_j do not appear in the where clause of H_i then $S_j+\Delta S_j$ can be replaced by 1 in T_i. As a consequence, if H_i has no where clause then T_i is replaced by K_i.

As an example of the way the failure counters work consider the following program which consists of four event handlers:

```
S, L1, L2, L3: single_use recurrent event ;

on program_start
   seq_cause L1; L2; L3; S
end ;

{ Event handler Hi (i=1,2,3) }
on Li ∧ S                            { a P operation }
   seq_cause ... ; S ; Li
end ;
```

The three event handlers H_1, H_2, H_3 correspond to three concurrent processes of the form:

```
Li:     P (S) ;                      { a P operation on S }
        ...
        V (S) ;                      { a V operation on S }
        goto Li
```

We can easily show that every event in event class L_i is eventually used; thus, none of the three corresponding processes can starve. It is not possible that a single EHM, say $M(H_i)$, always wins (succeeds to acquire an event from event class S each time such an event exists). The value of the counter C_{H_iS} associated with $M(H_i)$ remains constant whereas the values of the counters associated with the other two EHM's increase each time an event of class S is used. Eventually the value of the counter associated with another EHM becomes bigger than that of C_{H_iS}, and at that time $M(H_i)$ does not win but another EHM wins. Similarly, it is not possible that each time an EHM wins it is one of two EHM's, say $M(H_i)$ or $M(H_j)$, whereas the remaining EHM $M(H_k)$ never wins (i.e., starves). The reason is that the rate in which each of the two counters C_{H_iS} C_{H_jS} is incremented is less than once per use of an event from class S (since none of the two EHM's always wins as was shown earlier). On the other hand, C_{H_kS} is incremented by 1 after each use of an event from class S (since $M(H_k)$ always loses). Eventually the value of the counter C_{H_kS} becomes the biggest and $M(H_k)$ succeeds to acquire the next event from class S. The case in which none of the three EHM's ever wins is ruled out by our scheme since each time an event from event class S exists one of the three EHM's acquires it.

A nondeterministic state diagram can be drawn as shown in Figure 7.6. Each state in this diagram describes the values of the three counters C_{H_1S}, C_{H_2S}, C_{H_3S} relative to the value of the counter having the smallest value at that state. We assume the initial value of each of the three counters is 0. A transition from state S_j to state S_k in the diagram is labeled by the subscript i of the winner $M(H_i)$ in state S_j.

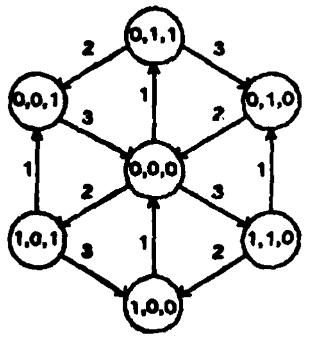


Figure 7.6 State diagram with failure counters

The intentional delay introduced by an ECM and the additional processing done by managers to achieve a fair implementation may cause performance degradation. The scheme can be improved. There are cases in which M(H) needs from M(E) only events satisfying some condition. For example, in the airline reservation system of chapter 6 an EHM may need the counter associated with a certain flight. In such cases, M(E) can remember the needed condition and introduce the delay period T' only when the appropriate condition is satisfied.

An EHM should repeatedly try to include each existing event in event collections it examines (unless it knows that the event cannot participate in any matching event collection). Otherwise, the scheme presented above does not guarantee fairness rules F1-F2. This requirement (from an EHM) does not pose serious difficulties. Each EHM should examine single_use events from each relevant event class E in the order in which they arrive to M(E) (unless E appears in a last predicate in the heading of H).

Events of multi_use types should be treated differently. Consider an event handler whose heading has the form:

on E_1 (...) $\wedge$ E_2 (...)

Suppose the types associated with E_1 and E_2 are single_use event and multi_use event respectively. If events from event class E_2 are examined in the order in which they arrive to M(E_2) it may happen that only the first event in the list is selected for matching event collections while other events are never selected.

In order to eliminate such anomalies, in each search performed by an EHM M(H) the first event to be examined from a multi_use event class E can be selected according to some round robin order (on the events of event class E). In addition, the multi_use event class whose events are examined first in the search can be selected according to some round robin order (on the multi_use event classes appearing in the event descriptor list of H). Finally, suppose H contains in its event descriptor list event class identifiers of multi_use and single_use event types. M(H) examines event collections by trying to match all possible combinations of single_use events to one combination of multi_use events, then iterates for another combination of multi_use events etc.

Note that the modifications in the implementation scheme suggested in this section do not introduce possibilities for deadlocks involving managers since timeouts are associated with the various waiting periods introduced above. The only negative effect of the modification is performance degradation; this is the price paid for achieving a fair implementation.

7.12 Optimizations

There are several sources for optimization in an EBL program. The simplest one is common subexpression elimination. This optimization can be applied both to the script of an event handler and to the boolean expression in the where clause of an event handler by well known techniques [Ah-77]. Other sources for optimization are derived from the unique semantics of the language and some of them are analyzed next. These optimizations should be viewed as compiler options which can be individually selected by the user. The degree of effort to be invested in each selected optimization can be another parameter specified by the user.

The basic event list organization suggested in previous sections was a doubly linked list ordered according to the arrival order of event objects to the ECM. However, there are cases which are easily detected by the compiler, in which this data structure can be either improved or a totally different organization selected. The optimization is based on the where clauses of event handlers containing the event class identifier in their headings, as well as on the type of the event class identifier.

7.12.1 Sorting an Event List

There are several reasons for sorting an event list. One reason, which has already been mentioned, is for preventing revisiting of an event collection in case of a single_use event handler with predicates. An event list can be sorted according to one or more keys (by creating indexes). If a where clause contains a min or max predicate whose argument is a function of the formal parameters associated with that event class identifier, sorting the event list according to that function can result in a better performance. The list itself can be sorted according to the function only if there is only one such function associated with the event class identifier, and the predicates first or last are not associated with the event class identifier in the whole program. If the list itself cannot be sorted, multiple indexes can be created which effectively sort the event list according to the functions.

What are the tradeoffs involved in sorting an event list? Suppose EHM M(H) wishes to find an event satisfying some min predicate from ECM M(E). Let n be the number of objects in the event list. In the most straightforward scheme, M(H) finds the desired object by sending n messages to M(E); each message requests the next object. The total number of messages is 2n. This scheme can be improved by allowing M(H) to request all objects in one message. The total number of messages in this case is reduced to n+1. A further reduction in the number of messages required can be achieved if M(E) itself finds the desired object. In this case, the total number of messages is reduced to 2, independently on n. The structure of the predicate need not be sent from M(H) to M(E); the compiler can supply the information to M(E). M(H) can be required at most to supply the values of some other events' parameters.

Thus, in order to reduce communication overhead an event list need not be sorted. The price for this reduction is paid by M(E) who has to perform more computation in order to find the desired object. The computation time is proportional to n, i.e., O(n). If the list is sorted the computation time (or more precisely the number of operations) is constant since the first object in the list is the needed one. If, however, the desired object has to satisfy some function in addition to the min predicate, as for example in:

on E_1 (i, j: **int**) $\wedge$ E_2 (k: **int**) **where** (j=k) $\wedge$ **min** (i)

the computation time in the worst case can be proportional to n, i.e., O(n). Note that inserting a new object into an event list requires an update of each index. The time complexity of this update is O(log n) for each index, if the index has a tree structure and the list contains n objects.

A totally different reason for sorting an event list arises in case of an event class identifier of a non_recurrent type. In this case, before inserting an event object into the list, the ECM has to verify that no identical element already exists in the list. The event list can be sorted (by containing an index when necessary) according to the event parameters. The time complexity of the search can be reduced from O(n) to O(log n) if the list (or the index) has a tree structure and the list contains n objects.

7.12.2 Preventing Re-evaluation of Expressions

The optimizations described earlier which prevent revisiting an event collection reduce significantly the amount of expression re-evaluation performed by an EHM. However, there are more cases in which optimization can be done. If the boolean expression in a where clause contains a subexpression which is a function of the formal parameters of one event descriptor (with corresponding event list k), this function can be evaluated once and stored as a hidden parameter of the event (this is a memoizing technique). The saving in computation time increases with the number of event descriptors in the event descriptor list, since the number of event collections for which the same value of the subexpression is needed increases. Let N_i be the number of existing events in the part of event list *i* to be searched by an EHM. The number of times the subexpression has to be evaluated can reach $\max\left(\prod_{i \neq k} N_i, 1\right)$ if the proposed optimization is not employed.

There is a time space tradeoff here. In our virtual system, the decision whether to apply this optimization is easy since memory is unbounded. On more constrained systems, the compiler must be given some information about the use of this optimization. There may be several degrees of this optimization. In the first one, it is not employed at all. In the second one, it is employed only for subexpressions of a boolean type (which waste very little storage). In the third one, the optimization is used wherever possible.

There may be several strategies regarding the evaluation time of the hidden parameters of an event. The simplest one is to evaluate when the event is caused. The drawback of this strategy is that if the event contains several hidden parameters for use by several EHM's, it may be that some of them will not be needed and computation time is wasted. A slightly complicated algorithm is to evaluate a hidden parameter the first time it is needed. This requires in general both storage to mark whether the parameter is already computed, and time for checking if it is already computed. These tradeoffs are reminiscent of those related to the various mechanisms for passing parameters to procedures; e.g., call by reference, call by name, call by value, or call by need.

7.12.3 Event Class as a Counter

In case of an event class identifier having no (explicit or implicit) parameters associated with it, the event list can be replaced by a counter counting the number of existing events from that class. The counter can only be incremented in case of a multi_use recurrent event, and can also be decremented in case of a single_use recurrent event. Accesses to the counter must be appropriately synchronized and this task can be achieved by the ECM. The space required to represent a list of n objects is constant (assuming overflow never occurs) as opposed to O(n) in the general implementation of an event list.

Suppose an event class identifier has parameters associated with it, and it does not appear as an argument in any of the predicates first, last, min, or max. If the total number of distinct parameter combinations that events from that class may assume is bounded by a small number k (such as in the case of the type **event** (**bool**, **bool**) for which k=4), the event list can be replaced by k counters, each corresponding to a specific parameter combination. The space required to represent a list of n objects is a constant O(k) as opposed to O(n) in the general implementation of an event list.

In all the above cases, if the event class identifier is of a non_recurrent type, each of the counters can be replaced by a single bit.

7.12.4 Event Class as a Record Variable

The problem of determining at compile time whether an event class can be represented as a record variable is an interesting one since such a knowledge can simplify the corresponding ECM. For an event class identifier of a single_use type it means to decide whether at any point in time at most one event object from that class exists. Unfortunately, since our language is universal, as can be seen from chapter 5, the above problem is equivalent to the halting problem.

In case of an event class identifier of a multi_use type, a similar decision problem exists. Since the semantics of multi_use events is that they are never forgotten, it seems that the prospects for optimization in this case are even fewer than in the previous case. However, there are cases which can be easily detected by the compiler in which an event class can be represented as a record variable.

One such case is that of an event class identifier E of a multi_use recurrent type such that the where clause of every event handler H containing it in its event descriptor list satisfies the following conditions:

1. No formal (explicit or implicit) parameter of the event descriptor associated with E appears in the boolean expression of the where clause.
2. E appears in exactly one event descriptor of H.
3. One of the following conditions holds:
    a. In every H, E appears as an argument of exactly one predicate first,

and no formal parameter of E appears in the argument of any other predicate.

b. In every H, E appears as an argument of exactly one predicate last, and no formal parameter of E appears in the argument of any other predicate.

The common characteristic of all the above cases is that the ECM must remember only one event object. The event object has some minimum (maximum) value associated with it: the index of the event object in the event list (which is not needed). In case a, only the first occurred event is remembered; in case b, only the last event which occurred so far is remembered.

Event objects which are no longer needed by the ECM cannot be simply garbage collected since references to them may exist in various instances of event handlers. A slight modification of the reference count mechanism (in which references within instances of event handlers are counted in addition to references within EHM's) can be used together with the state of a list element to appropriately solve the garbage collection problem in these cases.

7.12.5 Event Class as an Array of Records

The optimization discussed in the previous section can be extended to deal with an array of records if condition 1 is relaxed to the following one:

1'. Only formal parameters from a subset S of the set of formal parameters of E are used in [illegible]olean expressions in where clauses of all H.

In this case, the parameters identified by S can serve as array indexes. The array can be implemented as a sparse array since in general, not all its elements exist.

This optimization captures, among the others, uses of multi_use events for database applications as shown in the readers writers example in chapter 6.

7.12.6 Combined EHM's

A combined EHM can be associated with n>1 event handlers $H_1, \ldots, H_n$ containing identical event descriptor lists (except possibly different formal parameters), instead of n separate EHM's. The advantages of this approach are several: Less computational resources are needed, less messages are exchanged with the relevant ECM's, and the load on each of these ECM's is reduced since they have to handle fewer requests. Each of the above reductions can reach a factor of n (e.g., when the n event handlers are identical). One event space can be associated with the n event handlers; the combined EHM can cover it, find event collections matching one or more of the event handler headings, and activate instances of those event handlers according to our previous schemes.

If the event handlers are single_use event handlers, then once an event collection which matches one of them H_i is found, there is no need to check whether it matches the others. The reason is that after the acquisition algorithm is applied and terminates (successfully or unsuccessfully) the event collection no longer exists. If however, the event handlers are multi_use event handlers, checking the other event handlers is in general needed. The exceptions are those EHM's for which it has been established that the boolean expressions in their where clauses and the boolean expression of H_i are not mutually satisfiable.

As was indicated earlier, the problem whether two general boolean expressions, whose terms contain relations between integer expressions (containing the operators +, -, *, and /), are mutually satisfiable is undecidable. For simple boolean expressions in which the relations are restricted to atoms of the form:

<identifier> <relational_operator> <constant>

(for <relational_operator> as defined earlier in this chapter) the problem becomes decidable but *NP*-hard. Polynomial algorithms for more restricted cases, e.g., conjunction of atoms are known [Wo-77]. In such cases, the compiler can generate more efficient combined EHM's.

An example of a situation in which the conditions for creating a combined EHM are met is the equivalent of a case statement (or an if statement). In the general scheme described in previous sections, the conditions triggering each branch of a case statement are evaluated concurrently whereas here they are evaluated serially. This is the price paid for allocating fewer computational resources for this task.

7.12.7 Eliminating Redundant Events and Event Handlers

The first kind of redundant events are those which probably stem from programming errors. In one simple case, event class identifiers which do not appear in any event handler heading can be eliminated from the program (unless they are system event class identifiers) without changing the meaning of the program. The only changes in the program behavior are those related to execution times, over which the programmer has no direct control anyhow. In another simple case, an event handler containing in its event descriptor list at least one event class identifier whose events can never occur, can be eliminated. In a more general case, there is a group of event handlers $H_1, \ldots, H_n$ each containing in its event descriptor list at least one event class identifier from the set $E_1, \dots, E_m$ whose events can be caused only in $H_1, \dots, H_n$. In this case, all event handlers $H_1, \dots, H_n$ can be eliminated from the program without changing its behavior. We shall not pursue this kind of optimization since detecting such cases will normally cause the programmer to modify his program and recompile it.

A more interesting problem is that of detecting and eliminating intermediate events. This case is important since it occurs when a program is developed in a top down approach without deleting the intermediate steps (as discussed in chapter 3). The simplest case is that of an event class identifier E which appears in the event descriptor list of one event handler H; H contains only one event descriptor in its heading and has no where clause. In this case, E can be eliminated from the program. The script of H is appropriately substituted into every event handler where events from class E are caused. This substitution raises several problems which can be easily dealt with by the compiler, such as determining scopes of identifiers. Such a substitution may yield a script which defines a serial / parallel combination on its events (rather than the more restricted case of either a serial or a parallel order as allowed in EBL). Implementing these extended scripts should not pose serious problems.

In a slightly more general case, E appears in several event handlers $H_1, \dots, H_n$ each restricted as H above but a where clause without predicates is allowed in each H_i. A slightly more complicated substitution is needed here. It requires some kind of an if statement or a case statement within a script, and these can be easily implemented.

7.13 Tag Handling and Tag Optimizations

Tag handling without optimization is not complicated. A simple distributed tag allocation scheme allocates as a tag value some combination of the number of the processor which executes the instance of the event handler which defines the tag, and a unique number (unique to this processor). This approach eliminates the need for a central tag allocator. Tag operations are performed on those numbers in the obvious way.

Some optimizations can be performed by observing that the main (and intended) use of tags is for joining n concurrent branches of a computation. The general form of such a join is:

$$\textbf{on}\ E_1\ (\ \ldots,\ t_1\text{: tag},\ \ldots\ )\wedge E_2\ (\ \ldots,\ t_2\text{: tag},\ \ldots\ )\wedge \ldots \wedge E_n\ (\ \ldots,\ t_n\text{: tag},\ \ldots\ )$$
$$\textbf{where}\ \ B\ \text{and}\ t_1=t_2\ \text{and}\ \ldots\ \text{and}\ t_{n-1}=t_n$$

or slight variations of it, where B is a boolean expression (without predicates). A great inefficiency can result in cases where there are many events in each event class E_i, but the same tag value appears in few events; in particular, in exactly n events. This is normally the case when joining n branches of a computation whose instances are activated concurrently many times.

The feature of an ECM which allows specifying a simple condition in a *next* request saves communication overhead since the condition sent to E_i ($i>1$) can be $t_i=\alpha$, where α is the value of the tag parameter of E_1. However, each ECM has to scan its event list until an event with appropriate tag is selected.

An attractive optimization is to allocate a distinct processor each time a new tag value is needed and to take the processor number to be part of the tag value. This *tag processor* maintains a *tag list*. Each element of this list contains a reference to an event object carrying the same tag value and an identifier of the class of that event (e.g., the number of the processor on which the corresponding ECM resides). All events carrying the same tag are pointed to by the tag list.

Each time an event object carrying a tag parameter is inserted into an event list, a message is sent to the appropriate tag processor by the ECM; the tag processor inserts a new element into its list. Each time an event object is deleted, the corresponding element in the tag list is also deleted. The tag processor itself can be returned to the free processor pool when its list becomes empty and its tag value can no more be copied as a parameter of newer events (a state which should not be difficult to detect). This can happen only if no event object of a multi_use type is pointed to by the tag list, since multi_use events are never forgotten.

The overhead in finding matching event collections can be greatly reduced by using the tag processor. Instead of finding matching event collections by scanning event lists, it can be done by sending messages to the tag processor. In the normal (intended) tag uses, this approach decreases the time needed for finding a matching event collection. In other uses the time may increase. This situation can be easily remedied by using the original technique and the tag processor technique concurrently and aborting the losing one whenever a result is obtained.

7.14 Summary

A manager based implementation scheme has been developed in this chapter. This scheme associates an event class manager with each event class identifier in the program, and an event handler manager with each event handler. These communicating managers operate without any centralized control. A two phase distributed locking (acquisition) algorithm in which deadlocks are prevented has been developed. Many existing distributed locking algorithms prevent deadlocks by using a total order on all objects to be locked. Our locking algorithm only uses a partial order on all object classes. The advantage of this algorithm is that objects can be locked by a requestor concurrently (and not sequentially as in other algorithms).

The previous chapter has demonstrated the power and the roles of some of EBL's predicates. This chapter has shown that predicates somewhat complicate the implementation of the language. It seems that the complication is justified.

8. Network Implementation

This chapter investigates strategies for implementation of EBL on a processor network. In contrast to the previous chapter, this chapter does not assume unlimited computational resources. The network consists of a fixed number of processors p, each equipped with bounded local storage. For simplicity we assume that all processors are identical. The network is connected but each processor is directly connected only to a subset of the set of processors in the network, its neighbors. An example of a network satisfying the above conditions is the MuNet [Wa-78b]. We no longer assume that the cost of communication between program objects residing on different processors is zero. In fact, this cost will be one of the prime factors in the implementation scheme.

We would like to use the strategy outlined in the previous chapter but the limitations of the network pose several problems:

1. It may not be possible to allocate a distinct processor for each task; several tasks may have to share a single processor.
2. An object (e.g., an event list) may not fit into the memory of one processor and may spread over several processors.
3. Objects may have to move from one processor to another due to memory limitation in order to evenly share load among processors, or in order to decrease communication overhead.

8.1 Object Management

This section discusses the issues of object management and communication among objects in a network. The main kinds of objects in our scheme are: managers (ECM's and EHM's), event objects (which are kept within event lists managed by ECM's), scripts of event handlers (a copy of an event handler script may exist in more than one processor in the network), *temporary tasks* (instances of event handlers, or short tasks spawned by managers or by instances of event handlers), and messages exchanged among the above objects. An important issue in a network implementation, in contrast to the virtual system implementation of the previous chapter, is on which processor in a network an object resides. The issue is important due to its major effect on performance. A great deal of this chapter (sections 8.2-8.7) investigates the issue of finding good initial object distributions.

It is important to understand that once all managers are successfully allocated to processors (which must be done initially) there is never a necessity of moving a manager. If due to memory limitation something must be moved from one processor then temporary tasks, scripts of event handlers, and parts of event lists can be moved; the managers need not move. Moving a manager is only a matter of efficiency. The only reason for moving a manager is decreasing communication overhead. An ECM may move to processor P_u if most of its event list elements reside on P_u. An EHM M(H) communicating with several ECM's $M(E_1), \dots, M(E_n)$ may move to (or close to) $P(M(E_i))$ if the event list associated with E_i contains the largest number of events to be checked by M(H) over $E_1, \dots, E_n$.

Since objects may move among processors, the question of how they should be addressed arises. Our previous solution in which the address contains the processor number may not be efficient since each time the object moves references to it must be updated. In a more appropriate addressing scheme an object is referenced by using its name (a logical address). Some mechanism is needed for translating an object name to a physical (network) address. Such a translation can be achieved by keeping in each node a translation table. An entry in such a table can specify, for example, for each object name a processor number and an address local to that processor (or one of these two items and an indication whether the object is local or remote). This scheme is still not flexible enough (considering object movement) and suffers from a high space overhead.

For concreteness we will select a particular translation mechanism, although most of the discussions in this chapter are independent of this mechanism. For each object O_i a tree is created (*reference tree*) spanning all network nodes containing objects which reference O_i (as well as other nodes as needed for achieving a connected tree). This mechanism has been developed in [Ha-78, Ha-79]. Only a subset of the features of reference trees described there are needed in our implementation; however, the same name will be used here. In our case, a reference tree for object O_i is a rooted tree; the root is the node on which O_i resides. A principal advantage of reference trees is that if O_i moves, only the part of the tree to which O_i is connected need be updated, and not the whole tree. [Ha-78] describes a distributed algorithm for maintaining reference trees; a processor manipulates a reference tree only on the basis of interactions with its immediate neighbors in the network.

Other advantages are obtained if reference trees are used. Not all nodes in the network participate in each tree (i.e., not every node knows the names of all objects); therefore, space overhead can be smaller than in the case of translation tables (discussed earlier). Each node does not know the complete route to the root of the tree, but rather, only the next link to use (like in the Arpanet routing algorithm); this adds to the flexibility of the scheme. Routing of messages from a node on a tree to the root is trivial. The tree can also be used for communication of messages from the root to the rest of the tree as discussed next.

An interesting question is: in what ways can a broadcast from one manager to several others be implemented? The need arises when an ECM M wishes to broadcast the occurrence of some change in its event list to several EHM's. Similarly an EHM M may wish to broadcast to several ECM's a request such as "read boundary". One approach is that M sends separate messages, one for each destined manager along that manager's tree. The second approach is to broadcast the message on M's own tree from the root to the leaves. In every node to which the message arrives a check is made whether any manager is interested in the message. The message is passed on towards the leaves if there are more managers in that direction. Note that both approaches are possible since in our implementation scheme if manager M_i communicates with manager M_j then $P(M_i)$ is included in the tree of M_j and $P(M_j)$ is included in the tree of M_i.

The selection of the broadcast strategy can be dynamically made by M, based on information that it gathers. If M has to send messages to m managers $M_1, \dots, M_m$ and the distance between M and M_i along M_i's reference tree is D_i (assuming a distance 1 between neighboring processors) then the communication cost in the first approach is: $C_1 = \sum_i D_i$. In the second approach the communication cost C_2 is at most n-1 where n is the total number of nodes in M's reference tree. Keeping track of C_1 and C_2 is quite simple. C_2, for example, can be calculated by a distributed algorithm which propagates the appropriate information from the leaves towards the root of M's tree whenever changes in the structure of the tree occur. Since managers do not move frequently, the overhead involved in calculating C_1 and C_2 is expected to be low.

One approach could be to associate a reference tree with each object in the system. This approach may work but it totally ignores the special properties of the language. The advance knowledge of the kinds of objects dealt with and their relationships with each other can be used to decrease the number of needed trees by combining several trees into one (thus decreasing space and time overheads). In particular, there is no need to create a distinct tree for each event object. One reference tree can be associated with the corresponding ECM, and the ECM can maintain its objects independently of the reference tree by some less expensive mechanism. The problem of a list which does not fit into one processor can be dealt with by spreading the list among several processors and allocating secondary ECM's to these processors, controlled by the (main) ECM. A separate reference tree can join the secondary ECM's with the main ECM.

An ECM is a heavy object since in general moving it involves moving many objects (event objects). Moving a manager requires updating several manager trees; the tree associated with that manager, and those associated with the managers with which it communicates; as well as trees associated with temporary tasks. Thus, in our scheme managers will not move frequently, but rather, temporary tasks will be those which normally move.

Sections 8.2-8.7 investigate the problem of initial distribution of objects in a network. Section 8.8-8.9 investigate the problem of creating initial reference trees.

8.2 Initial Object Distribution

We could stop at this point and say that after a program is compiled the loader loads the objects generated by the compiler to one or more processors in the network regardless of their relationships with each other; during the course of the program execution objects will move in such a way that overhead is minimized and load is equally distributed. However, the transient period may last over most (or even all) of the program execution time. This is true in particular for programs whose run time is short.

Thus the problem of finding a good (or optimal) initial object distribution in the network according to some goodness criterion (which is discussed in section 8.3) is of great importance. Before a program is loaded, a global analysis can be made, based on data generated by the compiler and the current structure of the network, whose purpose is to find a good initial object distribution and the corresponding reference trees.

Problems similar to ours have been discussed in the literature. We shortly discuss some of them and explain why their results are not adequate for our purpose. [St-78a] discusses the problem of distributing program modules in a distributed system. As examples of distributed systems the paper gives: the Arpanet, C.mmp, Cm*, Pluribus, and a dual processor system. The program modules are assigned to processors in a way that minimizes a cost function Z. Z consists of the sum of communication costs for all pairs of communicating modules residing on different processors and the sum of execution cost for each program module at the processor on which it resides.

The paper implicitly assumes that the distance (or the delay) between every pair of processors in the system is identical (e.g., a fully connected network); this assumption is not correct for most of the above systems, and not in our case. The cost function does not make sense if the processors are identical since in such case its minimum is obtained by allocating all the program modules to one processor, ignoring the rest of the available resources, and that is not the paper's intent. In most of the above systems however, the processors are identical. The paper primarily deals with two and three processor systems, thus, the results are not applicable to our case.

[Je-77] deals with a similar problem but does not take into account storage cost, and as [St-78a] implicitly assumes equal distances between every pair of processors in the network. The method of finding the optimal distribution is simila. to that of [St-78a], finding cut sets in a weighted graph (the arcs are weighted) yielding a minimum total weight. The underlying model does not capture the problems we try to solve, therefore the results are not applicable to our case.

[Mo-77] deals with optimal distribution of programs and files in a network. The cost function consists of communication cost terms and storage cost terms. The paper assumes that program storage cost is negligible in comparison to file storage cost and solves the problem by decomposing it to individual file minimization problems. (The justification of the decomposition is given in another paper.)

The results cannot be used for our case due to several reasons. First, in our case the problem cannot be decomposed analogously to their case. Second, their storage cost is simply the sum of the storage cost of each file in the node on which it resides. If we use such a simple storage cost function in our case, then the storage cost function is constant for a given program since all processors are identical. Assigning all objects to a single processor will always result in minimum communication cost, and therefore in minimum total cost. Such a degenerate fixed solution is not acceptable in our case since it does not exploit available computational resources.

8.3 The Cost Function

The distribution problem is carefully analyzed in the following sections. The results are not limited to the context of EBL therefore the discussion proceeds in more general terms; the corresponding terms of EBL or our implementation scheme are given when necessary. We first define several concepts which are used in the sequel.

The *processor graph* PG (N, A) is an undirected connected simple graph representing the processor network. Each of the p processors is represented by a node in N, and each link is represented by an arc in A. A distance 1 is associated with each arc (assuming a constant delay between any directly connected processors). From the processor graph the *distance matrix* D can be constructed. The element D_{uv} of D represents the shortest path distance between nodes P_u and P_v in PG. Note that D is symmetric and $D_{uu}=0$ for all u. D can be constructed in polynomial time; an $O(p^3)$ algorithm which also produces the shortest paths (not only their lengths) is described in [Re-77b].

The *communication graph* CG (N', A') is an undirected (not necessarily connected) simple graph having n nodes representing the objects in the system and their interactions. A non-negative integer cost is associated with each arc in A', and a non-negative integer weight W_i is associated with each node in N'. The arcs costs can be represented by a symmetric *cost matrix* C. The element C_{ij} of C is the cost associated with the arc connecting nodes i and j. If there is no arc connecting nodes i and j then $C_{ij}=0$; $C_{ii}=0$ for all i since CG is a simple graph.

For each EBL program a communication graph can be created. The nodes are corresponding to the program objects whom we wish to distribute in the network (ECM's, EHM's, and scripts of event handlers). An arc between nodes i and j indicates that objects i and j communicate with each other. C_{ij} represents the communication cost per unit distance between objects i and j. W_i is the weight associated with object i; it may, for example, represent its storage requirement, its CPU requirement, or any combination of these two factors.

In the sequel, the following notation is used: n represents the number of nodes in CG; p represents the number of nodes in PG; i, j are nodes in CG; P_u, P_v are nodes in PG (however, they may also denote the corresponding processors in the processor network); P(i) is the node in PG to which object i is assigned (or the corresponding processor).

Our goodness criterion for a specific distribution will be the value of a cost function Z which we try to minimize. Z consists of two parts: Z_c the *communication cost*, and Z_l the *processor load cost*; $Z=Z_c+Z_l$. We assume that Z_c has the following form:

$$Z_c = (1/2) \sum_{i,j} C_{ij} D_{P(i)P(j)}$$

Note that if objects i and j are assigned to the same node in PG the contribution of their communication to Z_c is always zero. The processor load cost Z_l is the sum of the individual processor load cost Z_{lp} over all nodes in PG; i.e., $Z_l = \sum_{P_u} Z_{lp}$. There are many reasonable ways to select Z_{lp}. We shall not select a specific function; instead we assume that it satisfies the following conditions:

1. *weight*: Z_{lp} is a function of $\sum_i W_i$; where i ranges over the objects assigned to the particular node of PG.

2. *integrity*: $Z_{lp}(X)$ is defined for every non-negative integer, and assumes only non-negative integer values.

3. *threshold*: $Z_{lp}(X)=0$ for every non-negative integer $X \leq W_T$. The non-negative integer W_T is the threshold of the function.

4. *monotonicity*: $Z_{lp}(X+1) > Z_{lp}(X)$ for every integer $X \geq W_T$. This reflects that increasing the total weight on a processor (above the threshold) implies an increase in that processor's load cost.

5. *concavity*: $Z_{lp}(X) + Z_{lp}(X+2) \geq 2Z_{lp}(X+1)$ for every integer $X \geq W_T+1$. This guarantees that if the total weight at P_u is heigher than the total weight at P_v, moving an object from node P_u to node P_v does not increase the total processor load cost Z_l.

6. *polynomial*: $Z_{lp}(X)$ can be computed in deterministic polynomial time in the length of X.

Some examples of Z_{lp} are:

$$(1)\quad Z_{lp}(X) = \begin{cases} 0 & X \leq W_T \\ M+X-W_T & X > W_T \end{cases}$$

$$(2)\quad Z_{lp}(X) = \begin{cases} 0 & X \leq W_T \\ (X-W_T)M & X > W_T \end{cases}$$

$$(3)\quad Z_{lp}(X) = \begin{cases} 0 & X \leq W_T \\ (X-W_T)^2 & X > W_T \end{cases}$$

where W_T is a non-negative integer and M is a positive integer. In the sequel, a cost function means a specific cost function satisfying the previous conditions. The cost function may have several parameters: W_T is one parameter; M is an example of another one. Note that the evaluation of Z for a given object distribution can be done by a deterministic polynomial algorithm.

8.4 The Object Distribution Problem

In terms of the previously defined graphs, an *object distribution* is the association of each node of a given CG with exactly one node of a given PG. The problem we are trying to solve is the following.

The Distribution Problem:

Given a CG, a PG, a cost function, and the cost function parameters; determine the object distribution with minimum cost (the *optimal distribution*) Z_m. A closely related problem is the following.

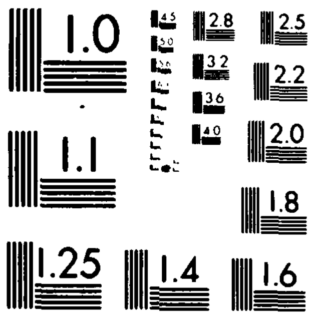
The Distribution Decision Problem:

Given a CG, a PG, a cost function, the cost function parameters, and a positive integer b; determine whether there exists an object distribution of cost at most b.

In practice, the object distribution may have to satisfy some constraints. An example of such a constraint is an object which must be assigned to a specific node of the PG; e.g., an ECM corresponding to a system event class identifier associated with some I/O device. Handling such constraints should not be difficult and we shall not treat them in the sequel.

AD-A081 950   MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 9/2
THE EVENT BASED LANGUAGE AND ITS MULTIPLE PROCESSOR IMPLEMENTAT--ETC(U)
JAN 80 A REUVENI   N00014-75-C-0661
UNCLASSIFIED   MIT/LCS/TR-226   NL

3 of 4
AD
A081950



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS 1963 A

Let us informally introduce several concepts which are used in this chapter; formal definitions can be found in [Ga-79]. *NP* is the class of all decision problems that can be solved by polynomial time nondeterministic algorithms. A decision problem P is *NP*-complete if it is in *NP*, and all other problems in *NP* are polynomially transformable to P. A problem P is *NP*-hard if there exists some *NP*-complete problem that is polynomially transformable to P. An *NP*-hard problem is at least as hard as any problem in *NP*.

Theorem 1

The distribution decision problem is *NP*-complete.

Proof: We first show that the Hamiltonian circuit problem (Does a p node undirected graph G have a Hamiltonian circuit?) is polynomially transformable to the distribution decision problem. We construct a PG G' by converting G to a graph satisfying the requirements of a PG. First, all parallel arcs but one connecting any two nodes, and all self loops are eliminated. Second, if G consists of q connected components then q-1 arcs are added to convert it to a connected graph. Note that G' has a Hamiltonian circuit iff G has a Hamiltonian circuit. As a CG we construct a ring of p nodes and p arcs. Each arc has a cost 1 associated with it, and each node has a weight W associated with it. W and the cost function parameters are selected in a way that guarantees that Z is minimized by an object distribution which associates exactly one node of the CG with each node of the PG. This can be easily done since $Z_c<p^2$. In particular, we choose $W=W_T=p^2$. Our transformation is clearly a polynomial transformation. In the optimal distribution, each node of the PG has a total weight W associated with it thus Z_l can be calculated; we get $Z_l=0$, therefore $Z_m=Z_c$.

It is clear that $Z_m=p$ iff G' has a Hamiltonian circuit, i.e., iff G has a Hamiltonian circuit.

To show that the problem is in *NP* we use the following algorithm which first chooses an object distribution:

```
for i=1 to n do                                  -O(n)
        $x_i \leftarrow$ choice ({1, ... , p})
end

if $Z(x_1, \dots, x_n, C, D) \leq b$                  -Polynomial
        then  success
        else  failure
```

where the meaning of the primitives choice, success, and failure is as defined in [Re-77b]. Since the above algorithm is clearly a nondeterministic polynomially bounded algorithm, the problem is in *NP*. This completes the proof of the theorem.

Corollary 1.1

The distribution problem is *NP*-hard.

We shall continue the analysis of the distribution problem in section 8.6. We now suggest several related interesting problems.

8.5 Network Design Problems

The problem of finding an optimal processor network configuration for a given communication graph G is an interesting one. If there are no limitations on the number of processors p and on the number of neighbors per processor one can build a processor network isomorphical to the communication graph or any network containing it. If a limitation on the number of neighbors per processor is added, the network described above can be modified by adding intermediate processors for increasing the number of links required by a processor beyond the limit. A more interesting problem is the one in which the total number of processors is limited to p. We shall define several related problems in this category and analyze them.

The Network Problem:

Given a CG, an integer $p \geq 1$, a cost function, and the cost function parameters; find a PG having p nodes such that the cost of the optimal distribution for that graph is the minimum among all possible PG's.

Solution: Finding the optimal distribution is not required in this problem. The optimal graph is clearly a complete graph of p nodes. (Not all problems are hard.) Note that the degree of each node in the PG is not explicitly bounded (although an implicit bound of p-1 exists). Problems in which the degree of each node is explicitly bounded are discussed shortly.

The Network-Distribution Problem:

Given a CG, a positive integer p, a cost function, and the cost function parameters; find a PG having at most p nodes and an object distribution for that PG such that the cost of this object distribution is the minimum among all PG's and all possible object distributions on them. The difference between this problem and the network problem is that here finding the object distribution is required. The following is a closely related problem.

The Network Decision Problem:

Given a CG, a positive integer p, a cost function, the cost function parameters, and a positive integer b; determine whether there exists a PG having at most p nodes such that the cost of the optimal distribution for that graph is at most b.

By adding to the above problems the additional constraint that the degree of each node in the PG can be at most $L \geq 2$ we get the following problems: the *limited neighbors network-distribution problem*, and the *limited neighbors network decision* *problem.*

Theorem 2

The network decision problem is *NP*-complete.

Proof: We first show that the partition problem (Given a sequence of n positive integers $S=(S_1, \dots, S_n)$ with sum 2K; is there a subsequence of S that sums to exactly K?) is polynomially transformable to the network decision problem. As a CG we construct a ring G of n nodes and n arcs. The weight associated with node i is $W_i=S_i(n+1)$. Each arc has a cost 1 associated with it. We choose p=2, b=n, and $W_T=K(n+1)$. This is clearly a polynomial transformation. If S has no partition then the cost of the optimal solution Z_m is at least n+1. This follows from the monotonicity of $Z_{lp}(X)$ and from the fact that the lowest possible increment in X is n+1.

$Z_m \leq n$ iff there is a subset of the nodes of G whose weights sum to exactly K(n+1); i.e., iff S has a partition.

To show that the problem is in *NP* we use the following algorithm:

1. { Choose a graph (define arcs) } $-O(p^2)$

```
for u=1 to p do
    arc_uu ← false
    for v=u+1 to p do
        arc_uv ← choice ({true, false})
        arc_vu ← arc_uv
    end
end
```

2. Prepare the distance matrix D $-O(p^3)$

3. { Choose an object distribution }

```
for i=1 to n do                                          -O(n)
        $x_i \leftarrow$ choice ({1, ... , p})
end

if $Z(x_1, \ldots, x_n, C, D) \leq b$                    -Polynomial
        then success
        else failure
```

Since the above algorithm is clearly a nondeterministic polynomially bounded algorithm, the problem is in *NP*. This completes the proof of the theorem. Note that the partition problem could also be used in proving theorem 1.

Corollary 2.1

The network-distribution problem is *NP*-hard.

Corollary 2.2

The limited neighbors network decision problem is *NP*-complete.

Corollary 2.3

The limited neighbors network-distribution problem is *NP*-hard.

8.6 Approximate Algorithms

The results of the previous sections motivated examination whether deterministic polynomial time approximate algorithms can be found to our optimization problems. First, let us define several terms (adapted from [Ho-78b]). Let Z_m be the cost of an optimal solution to an instance of an optimization problem P. Let Z^* be the cost of a solution to the same instance of P obtained by an approximation algorithm A. A is an *ε-approximate algorithm* for a problem P iff for every instance of P, $\frac{Z^* - Z_m}{Z_m} \leq \epsilon$ for some constant ε. An *ε-approximate* *problem* is a relaxed version of a given optimization problem P in which a solution given by an ϵ-approximate algorithm is required.

Theorem 3

The ϵ-approximate distribution problem is *NP*-hard for all $\epsilon>0$.

Proof: We show that the partition problem is polynomially transformable to the ϵ-approximate distribution problem. We use the construction in the proof of theorem 2. The weight associated with node i is $W_i=S_i(n+1)(1+\lceil\epsilon\rceil)$. Each arc has a cost 1 associated with it. We choose $p=2$ and $W_T=K(n+1)(1+\lceil\epsilon\rceil)$. Analogously to theorem 2, $Z_m \leq n$ iff the set S has a partition.

If S has no partition then the cost of the optimal solution Z_m is at least $(n+1)(1+\lceil\epsilon\rceil)$. This follows from the monotonicity of $Z_{lp}(X)$ and from the fact that the lowest possible increment in X is $(n+1)(1+\lceil\epsilon\rceil)$. If S has a partition but the approximate algorithm does not return the corresponding object distribution but a worse one, then the cost Z^a is again at least $(n+1)(1+\lceil\epsilon\rceil)$. This solution is not an ϵ-approximate solution since:

$$\frac{Z^a-Z_m}{Z_m} \geq \frac{(n+1)(1+\lceil\epsilon\rceil)-Z_m}{Z_m} > \frac{(n+1)(1+\lceil\epsilon\rceil)-(n+1)}{n+1} = \lceil\epsilon\rceil \geq \epsilon; \quad \text{i.e.,} \quad \frac{Z^a-Z_m}{Z_m} > \epsilon.$$

Therefore, the only solution approximating a solution with cost $Z_m \leq n$ (if S has a partition) also has the same cost. Thus, $Z^a \leq n$ iff the set S has a partition. This completes the proof of the theorem.

Theorem 4

The ϵ-approximate network-distribution problem is *NP*-hard for all $\epsilon>0$.

Proof: The proof is identical to that of theorem 3.

Corollary 4.1

The ϵ-approximate limited neighbors network-distribution problem is *NP*-hard for all $\epsilon>0$.

8.7 Heuristic Object Distribution Algorithms

In this section we resume the investigation of the distribution problem. The problem was shown to be *NP*-hard thus we resort to heuristic algorithms. We will present several algorithms finding good object distributions (although in general not the optimal distribution) and discuss their time complexity.

The common characteristic of all our algorithms is that they are two phase algorithms. The first phase consists of finding some initial object distribution. The second phase is identical in all the algorithms; it improves the initial distribution obtained by the first phase. We first describe the second phase algorithm, the *distribution improvement algorithm*.

8.7.1 The Distribution Improvement Algorithm

The algorithm improves a given object dist-ibution by trying transformations of limited types on the given distribution as long as the cost function can be decreased.

```
1.  Calculate Z, set change ← false.
2.  Search for the first object whose movement to another node in PG decreases Z.
        a.  If no such object is found set k ← 2 and goto step 3.
        b.  If an object is found, move it, calculate Z, set change ← true and goto
            step 2.
3.  If $K>K_m$ then goto step 5 else goto step 4.
```

4. Search for the first set of K objects whose interchange decreases Z.
   a. If no such set is found, set K $\leftarrow$ K+1 and goto step 3.
   b. If a set is found, interchange the objects, calculate Z, set change $\leftarrow$ true and goto step 4.
5. If change=true and $K_m>1$ then set change $\leftarrow$ false and goto step 2, else terminate.

The integer $K_m \geq 1$ is a parameter of the algorithm; the algorithm tries to interchange at most K_m objects in one step. K_m can be chosen according to the desired degree of effort after examining the time complexity of the algorithm. It can be chosen as a function of n and p such that the algorithm terminates in reasonable time. Since we only deal with integers and since each time the object distribution is improved Z decreases by at least 1, the algorithm is guaranteed to terminate after at most Z_o improvements; where Z_o is the initial cost computed at step 1. The time complexity for $K_m=1$ is determined by step 2 of the algorithm and is $O(Z_o np)$. The time complexity for $K_m>1$ is determined by steps 2 and 4 and is $O(Z_o n^{K_m}+Z_o np) = O(Z_o n(n^{K_m-1}+p))$.

Algorithms which try harder than the above one can be devised; e.g., instead of just interchanging up to K_m objects, moving up to K_m objects. However, the complexity of such algorithms cannot be justified especially since phase one of the algorithm is expected to generate good initial distributions.

We now present a sequence of increasingly improved (phase one) algorithms for generating the initial object distribution.

8.7.2 All in One (AIO)

This algorithm assigns all objects to one node of the PG. A reasonable choice for this node is the "center" of the network which can be found by calculating for each node in PG the function $L_u = \sum_v D_{uv}^2$. The center is the node having the minimum L_u. Obviously *unless processor load cost is negligible in comparison to communication cost*, the AIO algorithm presents the distribution improvement algorithm a bad initial distribution, thus forcing it to try hard (by choosing high enough K_m).

8.7.3 Best Fit (BF)

The BF algorithm first orders all objects in an *object list* OL in some random order. It starts with an empty distribution and assigns objects from OL to nodes of the PG one at a time according to their order in OL (left to right). After each step only a partial cost function Z^p can be computed since only some of the objects are assigned. (The components of Z^p are denoted by Z_c^p, Z_l^p, and Z_{lp}^p in the obvious manner.) At each step an object is assigned in a way which minimizes the increment ΔZ^p; i.e., the object is assigned to the best possible place (or one of them in case of ties). (The components of ΔZ^p are denoted in the obvious manner.) The first object in OL is assigned to the center of PG (as was previously defined).

The BF algorithm produces a better initial object distribution to the second phase than the AIO algorithm, but its weak point is the random order of the objects in OL; the next algorithm to be presented improves this point.

8.7.4 Spiral Best Fit (SBF)

The SBF algorithm is identical to the BF algorithm except that it does not order the objects in OL randomly but tries to order them in some promising way. The order is best explained in terms of an infinite tree (which need not be constructed) derived from the CG as follows. The first node of the tree is O_i such that $\sum_j C_{ij}$ is maximum over all i. In case of ties, one of the objects with maximum weight is selected. The sons of node j in the tree are all its immediate neighbors in the CG ordered by: (1) decreasing C_{jk}, then by (2) decreasing weight W_k. OL is then constructed from the tree by a breadth first order in which repetition is not allowed. Since the CG may be not connected, the process is repeated for all connected components of CG and the obtained lists are appended to OL at its end (the right side). The order in OL roughly corresponds to scanning the connected components of the CG one at a time, for each connected component in rings of increasing diameter centered around an initial node of the component; i.e., in a spiral course.

The SBF algorithm assigns groups of communicating objects, thus as long as processor load cost allows it such objects are expected to be assigned to close nodes of the CG.

8.7.5 Limited Search SBF (LSBF)

In each step of the SBF algorithm ΔZ^p is evaluated for all p possible assignments of the next object O_i. The LSBF algorithm detects cases which occur in every instance of the problem in which the search can be limited. Let us first define several terms. The neighbors of O_i in the CG which precede it in OL are called *left neighbors*. Node P_v of PG is said to be a *K-order neighbor* of node P_u if D_{uv}=K (K≥0).

In order to limit the search when assigning O_i the nodes of the PG are checked in a certain order. The first node to be checked is $P_u=P(O_j)$, where O_j is a left neighbor of O_i with maximum C_{ij} over all j. ΔZ^p is evaluated first for P_u, then repeatedly for K-order neighbors of P_u for K=1, ... In the worst case, all p nodes of the PG are checked. In the following cases, the search can be terminated earlier.

(1) If for some node P_v, $\Delta Z^p = 0$ then O_i can be assigned to P_v; no smaller change in Z^p can be found.

(2) If all left neighbors of O_i are assigned to one node (i.e., to P_u) and if assigning O_i to some node P_v results in $Z^p_{ip}=0$ for P_v, then the best assignment so far can be selected. The reason is that subsequently checked nodes cannot yield a smaller Z^p_c, and may yield a higher Z^p_i (in comparison to P_v).

(3) In this case (and the following one) we assume that $\sum_k C_{ik} > 0$ where k ranges over the left neighbors of O_i. Let M be a positive integer; and let S be the set of nodes in PG containing all K-order neighbors of P_u for K=0, ... , M. Assume that when checking all M-order neighbors of P_u the following conditions are satisfied:

a. All left neighbors of O_i are assigned to nodes in S; let these nodes be $P_1, \ldots, P_j$.

b. The assignment of O_i to each M-order neighbor of P_u results in $Z_{ip}=0$ for that node.

Let D_m be the maximum distance (according to the distance matrix D) between any M-order neighbor of P_u and any of $P_1, \ldots, P_j$. The search can always terminate after checking nodes up to and including all $M+D_m$-order neighbors of P_u, and the best assignment so far be selected. The reason is that for any M-order neighbor of P_u,

$$\Delta Z^p = \Delta Z^p_c \leq D_m \sum_k C_{ik}$$

where k ranges over all left neighbors of O_i. For L-order neighbors of P_u where $L > M + D_m$,

$$\Delta Z^p \geq \Delta Z^p_c > D_m \sum_k C_{ik}.$$

Checking the above conditions and keeping track of the needed values can be easily incorporated in the LSBF algorithm.

(4) The previous case can be generalized by modifying requirement b to the following one:

b'. The maximum ΔZ^p_{ip} resulting by assigning O_i to M-order neighbors of P_u is ΔZ_m.

Let $D'_m = D_m + \lceil \Delta Z_m / \sum_k C_{ik} \rceil$. The search can always terminate after checking nodes up to and including all $M+D'_m$-order neighbors of P_u and the best assignment so far be selected. The reason is that for any M-order neighbor of P_u,

$$\Delta Z^p = \Delta Z^p_c + \Delta Z^p_i \leq D_m \sum_k C_{ik} + \Delta Z_m \leq D'_m \sum_k C_{ik}$$

The rest of the argument is analogous to case (3).

8.7.6 Complexity of the Algorithms

The time complexity of the previously described phase one algorithms is as follows:

1. For AIO - O(n).
2. For BF - O(np).
3. For SBF the time of the initial ordering of the objects must be included. If the maximum degree of the CG is d (expected to be a small number for EBL programs) the ordering can be done by an O(nd log d) algorithm. The SBF algorithm is therefore of complexity O(np + nd log d) = O(n(p + d log d)). For most programs

and networks the ordering overhead is therefore negligible.

4. For LSBF the maximum complexity is as in the case of SBF, but the average complexity is expected to be smaller.

The complexity of the distribution improvement algorithm (the second phase) is the dominating factor in the distribution algorithm (on its two phases) for each of the suggested phase one algorithms. Thus, selecting a more complex phase one algorithm (e.g., SBF or LSBF) does not increase the total complexity. Moreover, since such algorithms are likely to present better distributions for the phase two algorithm, Z_0 is expected to be smaller in these cases, and thus the total complexity is expected to decrease.

8.8 Generating Initial Reference Trees

After the object distribution is found, reference trees should be created. Since in searching the optimal distribution shortest path distances between nodes in PG are used for calculating Z_c, the trees corresponding to the object distribution should be *shortest path trees*: the tree corresponding to object O_0, which connects $P(O_0)$ to $P(O_1)$, ... , $P(O_n)$ has the property that the tree path from $P(O_0)$ to $P(O_i)$ is a shortest path in PG for $i= 1, \ldots, n$. Shortest paths between all pairs of nodes in the PG are found while calculating the distance matrix D, and these paths can be used for generating shortest path trees. To create a shortest path tree connecting node P_0 with nodes P_1, ... , P_n the following algorithm can be used:

```
1.  Create an initial tree consisting of $P_0$.
2.  For i=1 to n
        a.  Select a shortest path connecting $P_0$ and $P_i$.
```

b. Starting at P_i towards P_o, add nodes and arcs from the selected path to the tree until a node already in the tree is encountered.

The fact that the reference trees corresponding to the initial object distribution are shortest path trees is important. If all objects are assigned to one processor at load time (an approach we have rejected) they spread throughout the computation but it seems that the resulting reference trees cannot be optimal since they are generated on the basis of local decisions. [Ha-78] describes trees which are far from being optimal that may be generated (although [Ha-79] describes better approaches).

In general, more than one shortest path may exist between two nodes in a graph, thus, when creating the shortest path tree we may have a choice. Is there a goodness criterion for choosing among several trees? The answer is positive; a tree having the minimum number of nodes is the optimal one since it involves the minimum number of processors. This is a second order optimization but let us investigate it. The problem we are trying to solve is the following.

The Shortest Path Tree Problem:

Given a PG G=(N, A), a distinct node P_o in N, and a set of nodes $P_1, \dots, P_n$ in N ($P_o \neq P_i$ for i=1, ... , n). The distance associated with each arc in A is 1 (according to the definition of a PG); find the *optimal shortest path tree* T; i.e., a tree connecting $P_o, P_1, \dots, P_n$ satisfying:

(1) The tree path connecting P_o and P_i is a shortest path in PG for i=1, ... , n.

(2) The number of nodes in T is minimum among all trees satisfying (1).

A closely related problem is the following.

The Shortest Path Tree Decision Problem:

Given the same information as in the shortest path tree problem, and a positive integer b; determine whether there exists a shortest path tree having at most b nodes.

By adding to the above problems the additional constraint that the degree of each node in the PG is at most L>3, we get the modified problems: the *limited neighbors shortest path tree problem*, and the *limited neighbors shortest path tree decision problem.*

Note that if PG is a tree, all above problems are trivial since there is only one tree connecting the desired nodes. The limited neighbors problems can be easily solved for L=2. In this case PG is either a tree (which can be drawn as a straight line) in which case the solution is trivial, or a ring in which case the solution is simple.

Theorem 5

The shortest path tree decision problem is *NP*-complete.

Proof: We first show that the set covering problem (Given a finite family of finite sets $S=\{S_j\}$, a positive integer K; is there a subfamily F of S $\{F_h\} \subseteq \{S_j\}$ such that $\cup F_h = \cup S_j$ (i.e., S has a cover F) and F contains at most K sets?) is polynomially transformable to the shortest path tree decision problem. As a PG we construct the following graph. The nodes of PG contain: a distinct node P_o, corresponding to each set S_j a node S'_j, and corresponding to each element Q_i in $\cup S_j$ (assume there are n elements in $\cup S_j$) a node P_i. The arcs of PG contain: exactly one arc connecting P_o and each S'_j; and exactly one arc connecting S'_j and P_i if Q_i is contained in S_j. The tree T should connect P_o with $P_1, \dots, P_n$; we choose b=K. This is clearly a polynomial transformation.

The number of nodes in T is at most n+1+K iff S has a cover F containing at most K sets.

To show that the problem is in *NP* we use the following algorithm which first selects a subgraph T of PG by choosing arcs a_i (among the r arcs of PG) and the nodes with which they are incident:

```
for i=1 to r do                                                        -O(r)
            $a_i$ ← choice ({true, false})
            { $a_i$=true means: arc i and the nodes with which it is incident are in T }
end

if T($a_1$, ... , $a_r$) is a shortest path tree connecting $P_0$ and $P_1$, ... , $P_n$ and
the number of nodes in T $\leq$ b                                      -Polynomial
            then success
            else failure
```

Since the above algorithm is clearly a nondeterministic polynomially bounded algorithm, the problem is in *NP*. This completes the proof of the theorem.

Corollary 5.1

The shortest path tree problem is *NP*-hard.

Theorem 6

The limited neighbors shortest path tree decision problem is *NP*-complete.

Proof: The proof is similar to that of theorem 5, thus only the required modifications will be given. We reduce the limited set covering problem (the set covering problem with the additional constraint that $|S_j| \leq 3$ for all j); this problem is *NP*-complete [Ga-79]. The construction is similar to that in the proof of theorem 5; however, intermediate nodes are added to satisfy the limited neighbors requirement; we choose L=4.

We first add nodes to the set of nodes $\{S'_j\}$ to complete $|\{S'_j\}|$ to the nearest higher power of two $M=2^m$ (such that $M \geq 2$); the resulted extended set is S''. We then add between node P_0 and nodes in S'' $M-2$ intermediate nodes that together with P_0 and the nodes in S'' form a constant height (m) rooted binary tree (the root is P_0) in which every intermediate node has two sons. We require that the tree T connects P_0 with the $M-2$ intermediate nodes (in addition to the nodes $P_1, \dots, P_n$).

An arc which previously went from node S'_j to node P_i now goes to an intermediate node P_{ij}. The graph under construction should connect node P_i with node P_{ij} for all j. The limited neighbors requirement is handled analogously to the way P_0 was handled. First, for each i, nodes are added to the set of nodes $\{P_{ij}\}$ to complete $|\{P_{ij}\}|$ to the nearest higher power of two N_i (such that $N_i \geq 2$). Then, a constant height binary tree is created whose root is P_i and whose leaves are the above N_i nodes. The height of this tree is $h_i = \log_2 N_i$ (where $h_i \geq 1$). This completes the construction which is clearly a polynomial transformation. Observe that the number of arcs on any shortest path connecting nodes P_0 and P_i is exactly $m+h_i$.

The number of nodes in T is at most $\sum_i h_i + (M-2)+1+k = \sum_i h_i + M-1+K$ iff S has a cover F containing at most K sets.

The other direction of the proof is identical to that of theorem 5. This completes the proof of the theorem.

Corollary 6.1

The limited neighbors shortest path tree problem is *NP*-hard.

8.9 A Heuristic Initial Tree Algorithm

This section presents a heuristic algorithm to the shortest path tree problem. The problem is a second degree optimization in the context in which it arises in our implementation scheme; thus, we will only describe a simple algorithm. First, assume that the algorithm for calculating the distance matrix D also produces for each pair of nodes in PG a list of all nodes included in at least one shortest path connecting them. Then, observe that the desired tree T is contained in the subgraph of G induced by the nodes P_o, P_1, ... , P_n and those in the lists associated with the pairs (P_o, P_1), ... , (P_o, P_n).

The algorithm operates on this subgraph G instead of on the whole PG. Let SP denote the set of nodes P_1, ... , P_n. The algorithm attaches a weight W_i to each node in G. If shortest paths from P_o to K distinct nodes in SP pass through node i whose distance from P_o is d, then W_i=(K-1)d; and the set of K points is denoted by SP_i. W_i represents the possible saving in the total number of nodes in the tree if the paths to the K nodes in SP_i coincide up to node i, in comparison to totally disjoint paths. The algorithm is the following:

1. If n=1 then goto step 4, else proceed.
2. For each node i in G evaluate W_i. Let node j be the node having the maximum weight associated with it.
3. If $W_j \neq 0$ then goto step 5, else proceed.
4. Here all shortest paths are disjoint. Return a tree containing an arbitrarily chosen path to each node in SP.
5. Recursively break the problem into two instances of the shortest path tree problem and merge the resulted trees. The two instances are:
    a. The distinguished node is P_o, the new set SP' is SP - SP_j U {node j}.

b. The distinguished node is node j, the new set SP' is SP_j - {node j}.

The algorithm always creates a shortest path tree, although not necessarily an optimal shortest path tree (as defined in the previous section). In order to see why the algorithm always creates a shortest path tree observe the following: For any two nodes A and B in an undirected connected graph, the length of the shortest path between nodes A and B is equal to the length of the shortest path between node A and node C plus the length of the shortest path between node C and node B, for any node C included in some shortest path between nodes A and B. The algorithm either returns a shortest path (in step 4), or breaks a path into two parts as described above (in step 5).

The algorithm seems to work fine on many toy examples since the weight function criterion captures nodes which are good candidates for being points at which a path splits to several paths. The algorithm fails in certain cases; one reason is the breaking of the problem into separately solved subproblems. This approach decreases the complexity of the subproblems therefore causes the algorithm to converge relatively fast; however, it lacks global overview and that is its main drawback.

The complexity of the algorithm is determined by step 2. Assume there are p' nodes in G (the subgraph of PG), then the time complexity of step 2 is O(np'). Note that $p' \geq n+1$; if $n=1$ then step 2 is not executed. The maximum number of instances of the problem generated by the algorithm for which step 2 is executed is at most n-1, thus the overall complexity is at most $O(n^2p')$. The average complexity is expected to be lower since when an instance of the problem results in $W_j=0$ no new instances of the problem are created.

8.10 Performance Evaluation

Appendix B develops bounds on the performance of a certain class of EBL programs on a network. We use the throughput of a program (the rate in which events from a distinguished event class are used) as a performance measure. The analysis assumes that various objects propagate in the network subject to constraints on link capacity and CPU capacity. The constraints and the function to be maximized define a linear programming problem whose solution yields the maximum possible throughput. The analysis is done first without assuming a specific implementation scheme, and then taking into account our manager based implementation scheme.

8.11 Summary

This chapter has further investigated the problem of implementing EBL on a processor network. A goodness criterion for the distribution of objects in a network has been developed, and several optimization problems involving the distribution of objects in a network has been analyzed. The problems, and even approximations to these problems, are *NP*-hard as we have proved in this chapter; several heuristic algorithms have been developed. The analyzed problems are not restricted to the context of EBL; they are of general interest.

9. Data Flow Implementation

This chapter investigates strategies for implementation of EBL on a data flow processor. A data flow processor (e.g., [De-77]) is designed to achieve a highly parallel operation and is therefore a natural candidate for implementing EBL which contains many sources for parallelism. A data flow processor is a special processor designed to run programs expressed as data flow schemata in a (graphic) data flow language. Its principal characteristic is that instructions are executed in response to the arrival of their operands and there is no notion of sequential control flow.

The architecture of a data flow processor is not unique; as a concrete example this chapter uses the architecture investigated at MIT/LCS Computation Structures Group. The architecture of the basic data flow processor as well as the various types of actors and arcs in a data flow schema are described in [De-77]. Figure 9.1 depicts the basic data flow processor.

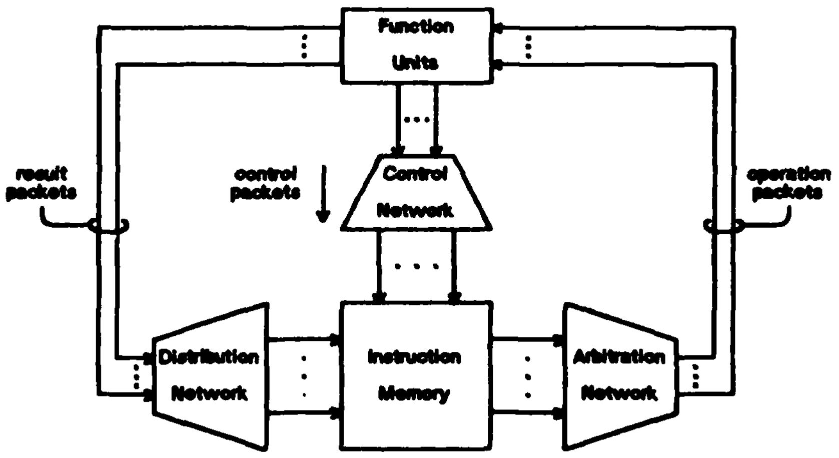


Figure 9.1 The basic data flow processor

Several extensions to this basic processor have been investigated: [Ac-77] describes adding a structure memory to the processor; [Mi-77] and [We-79] describe adding procedures to the processor. We will start with the basic data flow processor and examine the extensions needed in order to support EBL programs.

The discussions in this chapter will be conducted at two levels: the schema level, and the processor level. The first allows one to abstract out details, and the second explicitly deals with them. We assume that the reader is familiar with the main concepts of both levels.

Activating an instance of an event handler brings to mind activating an instance of a procedure. Activating concurrently several instances of an event handler is akin to activating concurrently several instances of a procedure. Unfortunately, the basic data flow processor does not support procedures. Since procedures and recursion can be easily expressed in EBL, any successful implementation of EBL on a data flow processor will actually add these features to the processor. Similarly, solutions to the procedure problem may make the effort of implementing EBL on a data flow processor easier.

Naturally, we have studied the existing proposals for adding procedures and recursion to the data flow processor [Mi-77, We-79]. The principal difficulty in implementing procedure calls on a basic data flow processor stems from the fact that the representation of programs in the processor is impure [Mi-77]; i.e., an actor and its input tokens are stored together in the instruction memory. Thus, in order to support concurrent activations of a procedure one cannot simply allocate a distinct working area for each instance of a procedure. A reasonable approach is to copy the procedure body (or the needed parts of it) for each activation of a procedure while relocating its addresses; relocating is needed in order to prevent undesired interactions between several instances of a procedure. This approach was adopted by both [Mi-77] and [We-79].

The common characteristic of both schemes is that they incrementally copy the needed parts for each activated instance of a procedure; thus, the cells associated with an unselected conditional branch are not copied. Both schemes use a virtual memory and some cache mechanism. [Mi-77] solves the relocating problem by using a centralized box which maintains a list of free unique identifiers. A unique identifier is associated with an instance of a procedure and is used as a suffix appended to cell names. [We-79] relies on the existence of a structure memory and uses unique identifiers whose maintenance is distributed in the processor and not centralized in contrast to [Mi-77]. For each procedure instance an activation record is created in the virtual address space; the starting address is based on a unique identifier. Both schemes also add special actors to the data flow language for supporting procedure activations. [Mi-77]'s scheme allows starting execution of the instance of the procedure even before all parameters are available. If we summarize the main mechanisms added to the basic processor by the two schemes we get the following:

1. A memory system in addition to the instruction memory.
2. Structure processing capability (in [We-79] only).
3. Cache mechanism (or memory hierarchy).
4. Address mapping (or relocating) mechanism.
5. Unique identifiers mechanism.
6. Special function units for handling procedure calls.

The above mechanisms are essential to the two schemes. The question arises whether one really has to add these mechanisms to the basic data flow processor and to copy a procedure body for each procedure activation in order to support procedures and recursion. The answer will be clear after our implementation schemes are presented in the following sections.

9.1 The Processor Architecture

The main effect of allowing recursive procedures in a language is that a program may use unbounded storage. This is one of the reasons why both [Mi-77] and [We-79] added a memory system (whose address space is much larger than that of the instruction memory) to the basic data flow processor. Since recursive procedures can be easily expressed in EBL, the need for such a memory system seems clear. From another point of view, an event list in EBL can grow beyond any bound, and this suggests that event lists be kept in a memory system which should be added to the basic data flow processor. Following this approach, the modified data flow processor assumed in this chapter will have the architecture depicted in Figure 9.2.

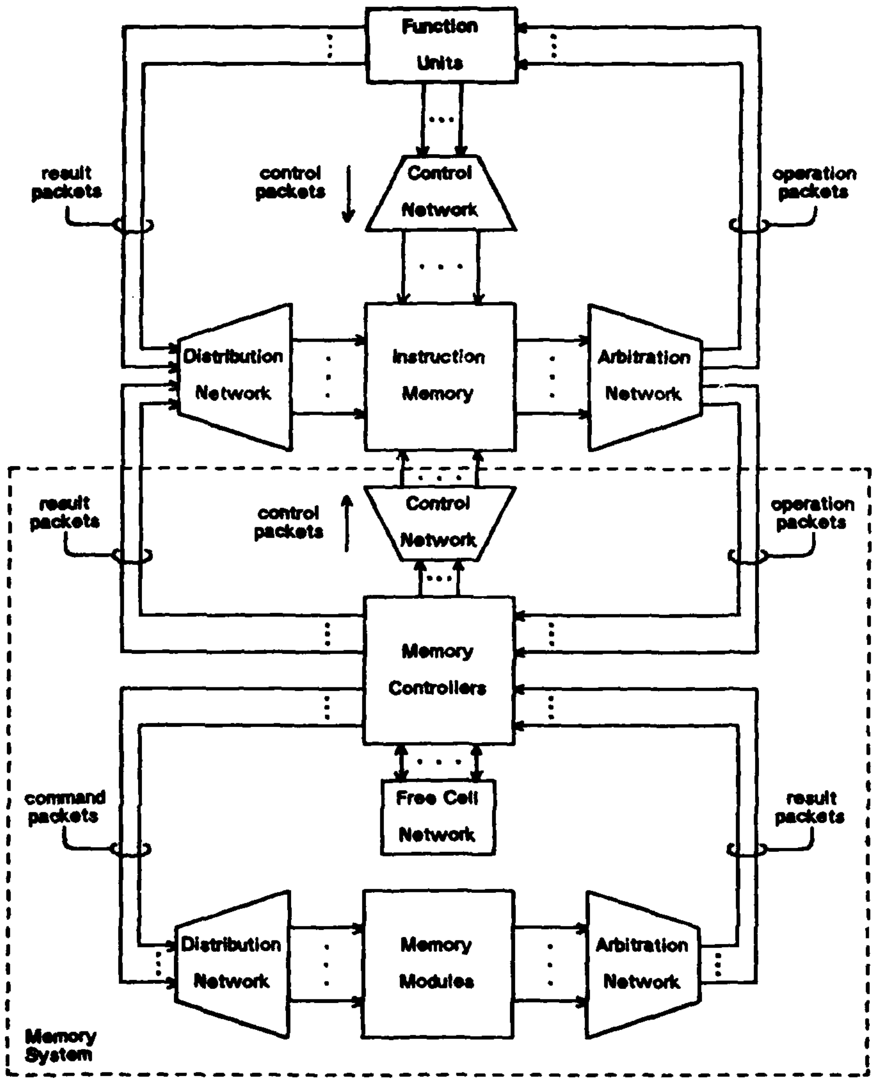


Figure 9.2 The basic data flow processor with the memory system

The memory system is described in detail in section 9.8; it is similar to the one suggested in [Ac-77]. An operation packet destined to the memory system arrives at any one of the identical memory controllers. Each memory controller can handle several operation packets concurrently. For each operation packet it normally exchanges one or more packets with the selected memory modules; it has access to all memory modules. Several result packets can be sent to the instruction memory in response to one operation packet. Packets generated in the memory system, carrying boolean values or signals, are sent through the control network analogously to the way such packets are handled in the basic processor; other packets generated in the memory system are sent through the distribution network. The free cell network supplies free storage cells to memory controllers. The memory controllers are "smart" controllers; they accept packets such as: write, read n words, insert a list element, delete a list element, get a free storage block, etc.

The following sections present several strategies for implementation of EBL on the data flow processor. The common characteristic of these strategies is that they basically follow the virtual system implementation scheme outlined in chapter 7; this common part of all implementation schemes is not repeated in this chapter. The main difference between the various strategies is the extent to which possible parallelism within a program is exploited; in particular, how many concurrent instances of an event handler are supported.

9.2 The One at a Time Scheme

This section presents a very simple operation mode of an EBL program. In this mode, at most one instance of each event handler is active at any point in time. Indeed, this mode does not meet our design goals since even the most straightforward form of parallelism in an EBL program is not exploited. However, this scheme exposes some of the problems posed by the underlying processor architecture and is upgraded in later sections. One must observe that this mode of cperation is semantically correct. Even though at any point in time several event collections may match an event handler heading, and thus more than one instance of that event handler could be active concurrently, an EHM can activate the instances one at a time; i.e., an instance of an event handler is not activated as long as a previous instance of that event handler is still active. The programmer has no way of forcing an EHM to activate more than one instance of the corresponding event handler concurrently. (In fact, even the more extreme mode in which at most one event handler instance is active at any point in time for the whole program is also possible; however, we shall not pursue this approach.)

In this scheme, each script and each manager are simply represented as data flow schemata containing several special actors. A script can be viewed as a single input single output schema. The input token, generated by the EHM, represents the activating event collection. The output token is a signal to the EHM indicating that this instance of the event handler has terminated. A script schema contains several kinds of actors.

1. Conventional actors: used for example for evaluating event parameters; these actors are implemented by function units.
2. Memory actors: used for example for reading parameters of activating events, for getting free storage blocks for preparing new event objects, and for writing parameters of new events; these actors are implemented by memory controllers.
3. Actors used for sending messages to ECM's.

Obtaining free storage blocks for events to be caused by an instance of an event handler need not be done by the instance of the event handler. The EHM can prepare the memory blocks in advance and pass them to the instance of the event handler when it is activated. This scheme decreases the execution time of the instance of the event handler but imposes an additional task on the EHM. If EHM's are not continuously busy, this scheme can decrease the execution time of the whole program.

A message can be implemented simply as a contiguous block either in the instruction memory, or in the memory system. Passing a message is reduced to passing a pointer to that block. The receiver decodes the message by examining its fields. The input to a script schema and messages to ECM's can be represented in this form. Actors for sending messages to ECM's are used both in scripts and in EHM's. The input to such an actor is a message containing a field defining the expected operation, as well as additional information depending on the requested operation.

When the requested operation is "add an event object to the event list" the additional information is a pointer to a memory block containing an event object which has been prepared by the instance of the event handler. The output ot the actor in this case is a signal token indicating that the actor's operation has been completed. As a side effect the actor normally adds the event object to the corresponding event list. If the event is of a non_recurrent type it may not be added to the event list; however, even in such a case the actor outputs a signal token. By defining the actor in this manner it becomes determinate, i.e., it outputs the same token when activated with the same input tokens. In fact, all actors in a script schema are determinate; this property is used in section 9.4.

In contrast to a script schema, a manager schema represents a cyclic process. This process receives requests from several sources; these requests are nondeterminately merged to one stream of requests and then manipulated. In order to implement a manager on a data flow processor one should first analyze several problems; for example:

1. How to merge requests from several sources?

2. Where and how to keep the state of a process?

The next section investigates these problems.

9.3 Some Basic Problems

This section analyzes several basic problems which are encountered not only in the one at a time scheme, but also in the more efficient ones. Similar problems are likely to occur in the implementation of many schemes involving concurrent processes on the data flow processor. This section suggests several modifications to the basic data flow processor.

9.3.1 The Merge Problem

The need to merge *messages from several sources* into one sequence of messages (i.e., to serialize the messages) arises in EBL in case of managers. For example, an ECM receives messages from EHM's and from instances of event handlers. There are several difficulties. First, a message may contain more than one field. Consider a naive solution in which a source sends each message field as a separate token to a fixed place in the schema which is determined by the message field and the receiver. In general, this solution does not work since the order of tokens in each of these places may not be the same. For example, in one place the token corresponding to a message from source A may precede the token corresponding to a message from source B, and in another place the reverse order may exist. The receiver which presumably takes one token from each of the two places may therefore process fields of different messages as one logical unit. Therefore, an undesired interaction of message fields (tokens) may occur. This difficulty can be easily dealt with by assuming that the message source (the sender) always prepares the message fields in a contiguous memory block and the message itself only contains a pointer to this block. The receiver, after obtaining a pointer accesses the various fields by using addresses relative to that pointer.

The second difficulty is that the sources may not be fixed, and the number of sources may vary and is not bounded in general. We first give a solution to the merge problem assuming fixed sources and then give a solution assuming varying sources.

Fixed Sources

Let us first deal with the case where there are n fixed sources $S_1, \ldots, S_n$. Assume that we have at our disposal a nondeterminate merge actor (see for example [We-79]) having n inputs. Can the schema of Figure 9.3 solve the problem?

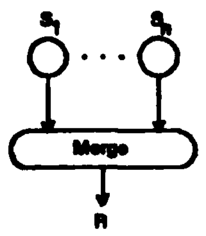


Figure 9.3 A nondeterminate merge

At the schema level the solution works fine. In its direct translation to a data flow processor program, the merging is obtained for free: the destination address in the instruction cell representing S_i is R for $i=1, \ldots, n$. Unfortunately, this direct translation may not work. The reason is that due to congestion of packets in the distribution network deadlocks (which are better called *congestion deadlocks*) can occur. The problem has been analyzed in [Mi-76]; the solution involves sending signal tokens as feedback from an actor to actors supplying its input tokens. Applying this idea, the schema of Figure 9.4 can be used to solve the problem.

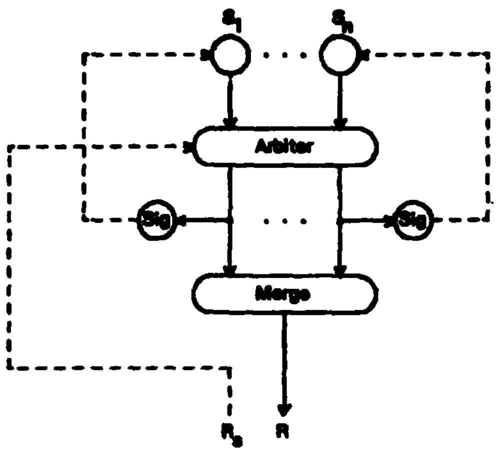


Figure 9.4 Safe nondeterminate merge

An *arbiter* arbitrarily chooses an input arc carrying a token and passes the token to the corresponding output when its signal input is enabled. When a *sig* actor fires a signal token is created; dashed lines represent arcs carrying signal tokens. When the receiver accepts a token at R, it returns a signal token at R_s by means of a *sig* actor (not shown in Figure 9.4). Note that here the operation of the merge actor can be really obtained for free. If the fan-in of actors is restricted (e.g., to 2), as is the case when one wants to represent each actor by at most one instruction cell, a schema for merging n sources can be easily constructed from limited fan-in merge subschemata.

Varying Sources

Here, the sources need not be fixed and the number of sources may vary and is not bounded. In this case, we would like to have a merge actor with variable number of inputs, with the ability of cancelling an input which will not be used any more by the source to which it was allocated; or the ability of reallocating an input to a new source. Without such abilities the number of inputs to the actor may grow beyond any bound even if the number of sources at any point in time is bounded by some fixed number. We do not have a direct way of implementing such a flexible actor on the data flow processor; instead, we use another approach.

Merging by simply having the same destination address at several cells (the sources) may cause congestion deadlock; our solution is to confine the congestion to a special area in the processor where it cannot cause the computation to halt due to deadlock. A new distribution network, called the *request network*, is added as shown in Figure 9.5.

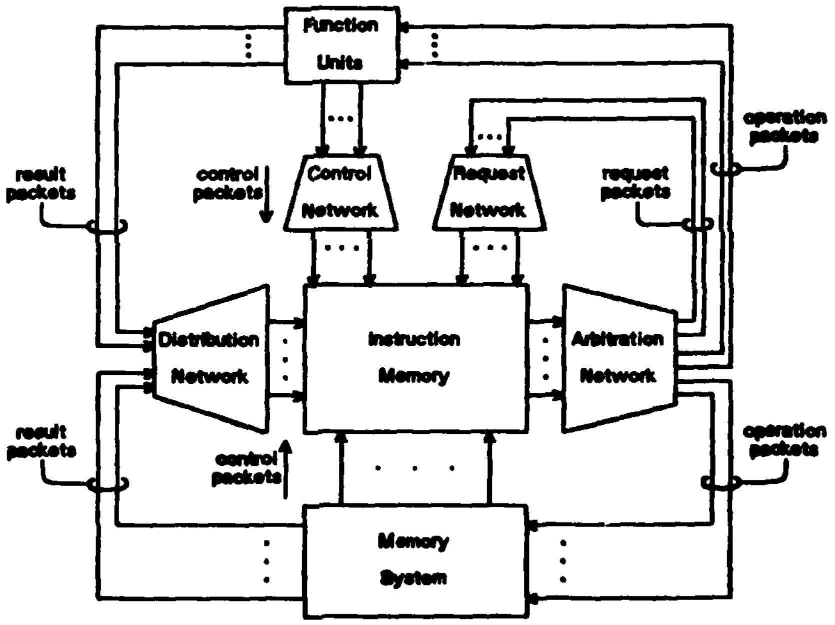


Figure 9.5 The data flow processor with the request network

In the new processor, in order to send a request to R a packet containing the request is sent to R; the packet is tagged as a *request packet*. Request packets are directed through the request network to their destinations. Congestions starting at the request network may propagate backwards to the arbitration network; in order to eliminate this, the arbitration network is split into two parts such that a congestion in the request network cannot cause a packet destined to a function unit or to the memory system to be blocked.

The request network provides a means for avoiding congestion deadlocks analogously to the way signals provide a means for avoiding congestion deadlocks; both do not guarantee deadlock freedom since they may be used improperly. In order to understand what is the proper use of the request network let us define several terms. We view a program as a collection of processes. A process can be cyclic or acyclic; it can be temporary or permanent. A packet from process P_i to process P_j can be a *request* R_{ij}; requests destined to P_j are merged within P_j. A request R_{ij} specifies some *processing* to be done by P_j. Throughout this processing replies may be sent to P_i. The *replies* are normal result packets or signals; however, requests sent to P_i are not considered to be replies. We assume that the message associated with a request packet contains addresses for sending the replies.

Any request R_{ij} whose blocking may cause P_j to stop processing other requests until R_{ij} arrives is a *critical request*. A critical request R_{ij} cannot be sent through the request network since it may be blocked by other requests destined to P_j or to other processes. Let us examine the following condition:

R1. For all i j, in order to process any request R_{ij}, P_j does not have to wait for another request destined to itself, or to send any request destined to another process and wait for any of its replies for completion of the processing of R_{ij}.

If R1 is satisfied then there are no critical requests and all requests can be sent through the request network; no deadlock will result from congestion in the request network. Of course, deadlocks due to congestions in the distribution network must be prevented; the scheme of [Mi-75] can be applied. Note that P_j is allowed to communicate with other processes while processing a request but not by sending requests for which replies are needed in order to complete the processing of the request. Also observe that R1 is a sufficient condition but not a necessary condition. We shall see (in section 9.7) that condition R1 is satisfied for the application of the problem in our implementation.

The request network can be more sophisticated than a regular distribution network. A packet can specify as its destination one of k adjacent addresses. The contents of these k destination cell registers can be merged according to the fixed sources solution to the merge problem. This approach associates a k word buffer with a receiver of requests and thus decreases congestion in the request network.

Another possibility for combining the fixed sources solution with the varying sources solution occurs when out of the requestors of P_j there are n fixed sources and the rest are varying sources. The fixed sources solution can be used for n+k sources, where k inputs to the arbiter are taken from the request network. This approach decreases congestion in the request network, and in addition, the n fixed sources can be served faster. Finally, note that the varying sources case of the merge problem occurs only in the script copying schemes discussed in section 9.7; for all other schemes suggested the fixed sources solution is sufficient and the request network is not needed.

9.3.2 The State of a Process

The basic data flow processor and the corresponding data flow language do not contain adequate primitives for handling side effects. In fact, the data flow methodology avoids side effects. One can therefore expect some difficulties in implementing variables; e.g., state variables of a process. Our implementation schemes involve processes (e.g., the managers) and this motivates our interest in the problem.

At the schema level, state variables of a process can be kept as tokens circulating in a cyclic subschema having the form of a feedback system [Le-79]; at the processor level, the state variables are kept in the instruction memory. In order to access a state variable from n different places in a process some interface should be added to the subschema; the size of this interface increases with n. In addition to the space overhead, as n increases a token trying to access the process state has to pass more actors, therefore the time overhead also increases with n.

Since we have added a memory to the processor we can use its cells to contain state variables in the conventional way. In order to access a state variable from n different places in the process only read or write actors are needed without any additional interfacing actors. Moreover, if a state variable is concurrently accessed from several parts of a program some synchronization mechanism is needed in general; a way to implement monitors in a data flow schema is described in [Le-79]. By devising powerful enough memory instructions of the flavor read-modify-write, the need for an additional synchronization mechanism can be eliminated in our scheme in many cases.

We next suggest by means of two simple examples several modifications to the data flow processor; these modifications make handling of state variables in the instruction memory more efficient.

9.3.3 The Alarm Problem

Suppose several sources in a program can detect alarm conditions. Each time an alarm is detected by some source, it should write the alarm status in a fixed alarm status word associated with the alarm handler process. The alarm status word is periodically checked by the alarm handler process which is only interested in the latest contents of the alarm status word. The problem arises in our implementation schemes when several ECM's signal an EHM that it can begin a search for matching event collections. One can view the alarm status word as a state variable and treat it in one of the ways suggested earlier, but more efficient schemes can be devised. Consider the schema in Figure 9.6.

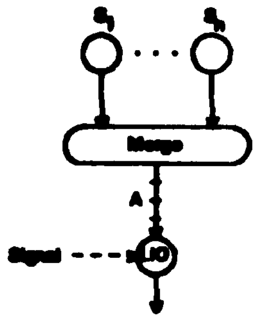


Figure 9.6 Last in out

Arc A is a special arc of unlimited capacity. It allows the actor of which it is an output arc to fire even if it carries tokens (in contrast to the normal firing rule). The order of tokens on A is preserved. *LIO* is a special actor: when a signal token arrives at its signal input it waits until there is at least one token on its input arc; it then swallows all existing tokens on that arc and outputs only the latest one. This is not a LIFO (last in first out) actor, but a LIO (last in out) actor.

The schema of Figure 9.6 exactly solves the alarm problem; how can it be translated to the data flow processor? The merging is achieved implicitly as was shown. The function of the LIO actor can be obtained by allowing an instruction cell register to operate in the following mode:

1. The register is enabled after receiving at least one packet.
2. The register is ready to accept packets addressed to it as long as the instruction cell itself is not enabled. The latest accepted packet determines the current contents of the register.
3. The register is reset after an operation packet leaves the enabled instruction cell to the arbitration network (i.e., the cell fires).

9.3.4 The Multi-State Process Problem

Suppose a process (e.g., an EHM) handles its input tokens according to the current state of its state variables; in particular, assume that the output of actor P should be directed to either actor Q or actor R depending on the state of the process as can be seen in Figure 9.7.

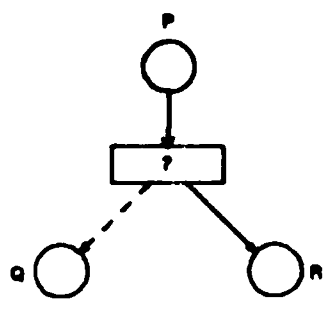


Figure 9.7 Switching an actor's output

How can this behavior be achieved on the data flow processor? The problem can be treated as a regular problem involving a process state as was discussed earlier. The state of the process should be checked for each token output by P. If the state changes infrequently relative to the frequency in which tokens are output by P, or if the decision where to send the output of P changes infrequently, then there is redundant token movement in the schema, or excessive packet traffic in the data flow processor.

An attractive way is to allow dynamic modification of arcs in a data flow schema. One can specify that whenever an arc moves, tokens residing on it move with it. This however, leads to nondeterminate computations and there is no need to add another source of nondeterminacy to our language. Thus, we shall assume that whenever an arc moves it carries no tokens. This approach amounts to dynamic modification of a program. In general this is a dangerous approach, but if it is done in a controlled and tested manner by system programs or by output of system programs such as compilers (in particular, an EBL compiler), it can be quite useful. In order to see how this can be implemented in the data flow processor let us examine Figure 9.8.

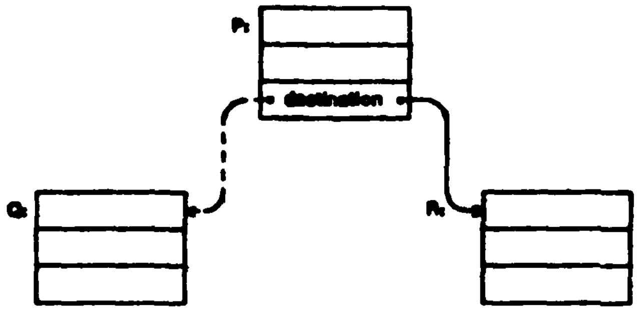


Figure 9.8 Instruction cell representation

From Figure 9.8 it is clear that switching the output arc of actor P simply means changing the destination field in instruction cell P. Note that the assumption that the arc connecting P and R carries no tokens means that the input register of R is empty, there is currently no packet in the processor whose destination is the input register of R, and there is no packet destined to a function unit or to the memory system which can yield such a packet.

Allowing packets to be sent to the register containing the destination address of a cell is a simple task; however, this means that the register becomes a variable register. Such a register is normally reset after the contents of the instruction cell is sent to the arbitration network [De-77]. This may force us to load (send a packet to) the destination address register of cell P before each firing. This undesired behavior can be eliminated by allowing an instruction cell register to operate in the following mode:

1. The register is enabled after receiving at least one packet.
2. The register is ready to accept packets addressed to it as long as the instruction cell itself is not enabled. The latest accepted packet determines the current contents of the register.
3. The register is not reset after an operation packet leaves the enabled instruction cell to the arbitration network.

9.3.5 Instruction Cell Modifications

The modifications of the data flow processor suggested in the previous two subsections can be generalized by allowing the independent definition of the following orthogonal modes for every field in an instruction cell:

1. *Reset mode*: When off, the cell field is not reset after a cell firing and the latest contents is used. When on, the cell field is reset after each cell firing and needs

a new packet in order to be enabled again (as an option, several new packets can be specified instead of one).

2. *Ready mode*: When off, the cell field does not accept a packet when the field is not empty (this is the current mode in the data flow processor). When on, the cell field is ready to accept packets addressed to it as long as the instruction cell is not enabled. The latest accepted packet determines the current contents of the cell field.

It is important to understand the effect of the modifications suggested above. The data flow processor has primitives which allow data flow schemata to be translated to it. However, it is more general than the schemata in the sense that computations which cannot be expressed as data flow schemata can be represented on it. As an example, suppose the only available actors are identity actors. The schema in Figure 9.9 represents a FIFO of capacity two.

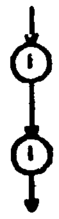


Figure 9.9 A FIFO

Figure 9.10 depicts the "equivalent" program obtained by a direct translation.

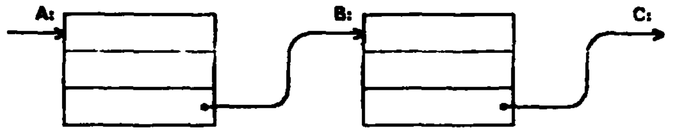


Figure 9.10 Cell representation of a FIFO

The behavior of the translated program is nondeterminate in contrast to that of the schema. The reason is that the order of the packets arriving at A is not guaranteed to be preserved at C. Moreover, the translated program may deadlock in contrast to the original schema.

The point we try to make is that the primitives of the data flow processor allow representation of a richer class of computational behaviors than data flow schemata. The modifications suggested earlier increase the above difference and allow better performance.

9.4 The Pipeline Scheme

In the one at a time scheme discussed earlier, an activation of an instance of an event handler has to wait for the termination of the execution of the previous instance of that event handler. This section shows that there is no need to wait and the execution of several instances of the same event handler can proceed concurrently in pipeline mode. In order to understand why this can be done consider a data flow schema S satisfying the following condition:

P1. S is a single input acyclic data flow schema which does not contain nondeterminate actors, conditional actors, or procedure activations.

For each activation of S all actors of S fire, and each actor fires exactly once. The firing rules for a data flow schema are such that if several activations of S are executed concurrently, then tokens belonging to different activations do not interact with each other.

Furthermore, if token t_j precedes token t_k at the input of S, then for each actor A_i having n output arcs in S the output token n-tuple corresponding to t_j $T_i(t_j)$ precedes the output token n-tuple corresponding to t_k $T_i(t_k)$ in the sequence of token n-tuples output by A_i; i.e., the tokens order is preserved throughout the whole computation.

Condition P1 can be relaxed by allowing conditional actors (such as a switch actor [We-79]) as long as the schema is a well formed acyclic data flow schema (as defined for example in [De-73]). In this relaxed case, not all actors of S must fire for each activation of S. Tokens belonging to different activations of S do not interact with each other because the schema is well formed and acyclic. If token t_j precedes token t_k at the input of S, then for each actor A_i in S if $T_i(t_j)$ and $T_i(t_k)$ exist then $T_i(t_j)$ precedes $T_i(t_k)$ in the sequence of token tuples output by A_i. Note that in this case, if $T_i(t_j)$ exists then $T_i(t_j)$ is a k-tuple for some $0<k\leq n$; for a switch actor n=2 and k=1. We shall not pursue the use of conditional actors any further since our event handler scripts contain no conditional elements.

Condition P1 can be extended to cover the case of a schema with more than one input:

P2. S is an n input acyclic data flow schema which does not contain nondeterminate actors, conditional actors, or procedure activations.

S is activated for logically related token n-tuples. Let t_{ij} denote the token at input i $(1\leq i\leq n)$ of S belonging to token n-tuple j. In order to use a general schema S satisfying condition P2 in pipeline mode the following condition must be satisfied:

P3. t_{ij} precedes t_{ik} in the sequence of tokens input by S at input i, implies t_{mj} precedes t_{mk} in the sequence of tokens input by S at input m for all possible i, j, k, m.

If condition P3 is not satisfied, undesired interactions among tokens belonging to different logical input token n-tuples may occur.

Condition P3 may be difficult to achieve; for example, when different asynchronous processes concurrently activate S. Our solution to this synchronization problem is to transform the schema S having n>1 inputs to a schema S' having one input by passing a pointer to a contiguous block containing the n original inputs. This approach was used earlier in the merge problem. If however, condition P3 can be guaranteed, the conversion to S' is not needed. The advantages of using S over S' are that S contains fewer actors (less space) and is executed faster. Faster execution is achieved since packaging of the inputs into a block and extracting them from the block are not needed; and execution of S can start before all n inputs are available. One case in which condition P3 can be easily achieved is when a single process creates the inputs of S.

An EBL script can be represented as a data flow schema whose inputs are pointers to n event objects forming an event collection. From the syntax of an EBL script it can be seen that condition P2 is satisfied for every script. Condition P3 is also satisfied since a single process (the relevant EHM) activates instances of the script. Thus, several instances of an event handler can be executed concurrently in a pipeline mode.

Note that before a schema S is translated for the data flow processor it should be transformed to a safe schema, as described in [MI-75], in order to avoid congestion deadlocks. Such a transformation is not needed for event handler scripts in the one at a time scheme. When the pipeline scheme is used, this transformation is required and can be done by the compiler. All known results about pipeline techniques can be applied to our scheme. For example, in order to increase the throughput of an event handler (the number of instances activated per unit time), identity actors can be introduced in some branches of the schema; this however, increases the number of packets sent per execution of an instance of an event handler.

9.5 The Multiple Script Copies Schemes

In the two schemes described in the previous sections exactly one copy of a script of an event handler was stored in the data flow processor. Each of these schemes can be modified by having $n \geq 1$ copies of a script; n is fixed for each event handler. By extending the one at a time scheme, we get a scheme in which at most n instances of an event handler can be concurrently active, the *n at a time scheme*. By extending the pipeline scheme, we get a scheme in which at most n pipelines can be concurrently active, the *n pipelines scheme*. In both schemes, the EHM can start activating a new instance of the event handler when there is a *free script copy*. In the n at a time scheme, a script copy becomes free once its execution has terminated (all actors have fired). In the n pipelines scheme, a script copy becomes free after a signal has arrived from each of its input actors (or in another approach from at least one of its input actors) to the EHM.

For both schemes, in the simplest approach the EHM uses the script copies on a round robin basis; the next script copy to be used is always known, and is used only after it becomes free. This approach may cause unnecessary delays since script copies may not become free in the order in which they are used. In an improved scheme, the EHM maintains a list of free script copies; the least recently used free script copy is selected by the EHM.

The fact that the script copies are fixed implies that each ECM can use the fixed sources solution to the merge problem. In particular, this means that the request network need not be added to the processor and the architecture of Figure 9.2 can be used.

A compiler cannot make an intelligent selection of n for each event handler just by analyzing a program. In general, such a selection requires understanding of the program and knowledge of the values of run time parameters or ranges of such values. Therefore, a reasonable approach is to let the user specify n for each event handler. The specification can be done within the program text itself or by some compile time dialog. By allowing the user to arbitrarily select n, even the smallest program may need unbounded space for storing its code (the script copies). Unless virtual memory is added to the processor, it may be necessary to eliminate script copies at load time in order to meet the current available space in the instruction memory. Again, some load time dialog can be imagined in which the user incrementally decreases n for various event handlers until the complete program can be loaded.

9.6 The Virtual Memory Fixed Schemes

All schemes presented so far implicitly assume that the instruction memory can contain all managers and all script copies together. The assumption may not be valid for large programs, for programs where many copies of scripts are desired in order to obtain higher concurrency, for processors containing a small instruction memory, or when more than one of these cases occur. A natural solution to the problem of a too small instruction memory in such cases is to use a sufficiently large virtual memory. The virtual memory can be implemented as a hierarchy of memories; for example, as a two level memory. The instruction memory serves as the cache, and some packet memory serves as the second storage level. This modification of the data flow processor allows all our previous schemes to be used without any change, assuming that the cache mechanism is transparent to the program; the number of script copies and their virtual addresses are fixed, hence the name of the schemes outlined in this section.

Virtual memories and cache mechanisms are described in [MI-77] and [We-79] as parts of their schemes for adding procedures to the data flow processor, and in [Ac-77]; similar mechanisms can be used in our case. Only the virtual memory and cache mechanism of [MI-77] or [We-79] are needed; and not the additional mechanisms supporting procedure calls such as: unique identifiers, or special function units for handling procedure calls.

The existence of a virtual memory allows one to use a big number of script copies in the multiple script copies schemes in order to increase concurrency. However, using too many copies of the same script may cause unnecessary trashing. If the replacement algorithm used by the cache mechanism is known to the compiler, an EHM can select the next free script copy in a way which minimizes the likelihood of a cache miss. Suppose for example that the least recently used (LRU) replacement algorithm is used. Selecting for the next free copy to be used the least recently used free script copy as suggested in the previous section is the worst approach when trashing occurs; although it is the best approach for trying to keep all n copies of a script in the cache. Selecting the most recently used free script copy is a better approach for reducing trashing. Generalizing from this (LRU algorithm) example, it seems that the criterion used by an EHM for selecting the next free script copy should be the inverse of the criterion used by the replacement algorithm.

9.7 The Script Copying Schemes

We stated earlier that solutions to the procedure problem can be used in the implementation of EBL. The main characteristics of the two currently existing proposals for solving the problem [Mi-77, We-79] have been described. This section demonstrates how Miranker's scheme ([Mi-77]) can be used. The idea is quite simple: we view the script of an event handler as a procedure P, and the activation of an instance of the event handler as a call to procedure P. Miranker's scheme cannot be employed in every case since unsafe conditions may occur [Mi-77]. Thus, we first have to demonstrate that the use of the scheme in the context of our implementation is safe.

At some level of abstraction an EHM can be viewed as a loop containing a call to procedure P. A potential source for unsafety in such activation is that several concurrent instances of P may return result to the same destination. This cannot occur in our implementation since P returns no values to the calling EHM. According to Miranker, a sufficient condition to ensure correct and safe operation of his scheme in this case is that all input values for a given activation of a procedure arrive before any of the input values for the next activation [Mi-77]. Since all activations of P occur within the EHM, the above condition can be easily guaranteed.

The special properties of our event handlers can be used to simplify Miranker's scheme. In his scheme, a special actor RET is used to support returning of a value from a procedure to its caller. Since no values are returned in our case, this actor is not needed. A special actor FREE is activated at the end of the execution of an instance of a procedure with which a suffix σ has been associated. Its tasks are purging all instruction memory cells having suffix σ, destroying all packets with a name with suffix σ, and returning σ to the list of free unique identifiers (suffixes). In our case, every actor of the procedure fires exactly once; therefore, each procedure cell can be purged immediately when it fires. This approach is reminiscent of incremental garbage collection. In our case, this incremental garbage collection is simpler than the garbage collection needed in Miranker's scheme and provides better space utilization. Returning σ to the list of free unique identifiers can be done at the end of the execution of the instance of the procedure as in [Mi-77].

Another simplification, also based on the fact that each actor in our procedures fires exactly once, can be made. There is no need to convert a procedure body to a safe schema as described in [Mi-75] in order to prevent congestion deadlocks. This simplification results in better space utilization and smaller communication overhead.

An interesting observation is that in Miranker's scheme if a suffix σ is limited to n bits then at most $N=2^n$ procedure calls can be executed concurrently. This limitation not only restricts the degree of concurrency but also the computations which can be performed on the processor. For example, factorial(N+1) cannot be computed by the conventional recursive program. Using Miranker's technique in our scheme does not restrict the possible computations. When recursive procedures are expressed in EBL, the state of the computation at each instance of a procedure is kept in one or more event objects and these have nothing to do with suffixes.

The schemes of [Mi-77] and [We-79] incrementally copy instructions constituting the procedure body as they are needed. Another strategy is to copy the whole procedure body at call time. The time space tradeoffs of the two approaches have been discussed in [We-79] and will not be repeated here. In our case, there is additional argument in favor of the latter approach: the fact that all actors of the procedure fire and no instructions are fetched in vain (as happens when the procedure contains conditional actors). Copying a whole procedure is simple. In the case of [Mi-77] it can be easily achieved if a procedure occupies a contiguous area in the virtual memory; the compiler can guarantee such a memory allocation.

The idea of copying a whole procedure body can be pursued to further specialize the schemes of [Mi-77] or [We-79] to our needs. Since in our case each procedure has exactly one caller (the EHM), this caller can explicitly initiate the copying mechanism. Furthermore, each EHM can pre-copy several instances of a script and select one of these ready script copies when needed. This approach can improve the time performance unless too much thrashing occurs.

All the schemes described in this chapter (except in this section) did not require the use of the request network. However, in order to use the schemes of [Mi-77] or [We-79] the request network is needed since the sources generating requests to an ECM are varying. Condition R1 of section 9.3.1 is satisfied since each request arriving to an ECM can be individually processed without the need to send or receive other requests. Thus, all requests to ECM's can be passed through the request network. As suggested at the end of section 9.3.1 requests from EHM's to ECM's can be directed through the distribution network in order to serve them faster and to decrease congestion in the request network; this can be done since the EHM's are fixed sources.

9.8 The Memory System

The architecture of the memory system depicted in Figure 9.2 has been briefly described earlier. It closely resembles the structure memory described in [Ac-77] and our memory system is an adaptation of the structure memory. The structure memory as described in [Ac-77] does not support side effects; in fact, avoiding side effects is one of its prime goals. For example, if two pointers P_1 P_2 in the processor point to the same structure S, and one of these pointers P_1 is involved in an operation which modifies a component of the pointed structure the structure pointed to by P_2 is not affected by the operation. Conceptually, the operation creates a new structure although in the implementation parts of the two structures are physically shared.

If in the previous example one wants that P_2 will reflect the operation performed on the structure S; i.e., will point to the modified structure, it cannot be achieved. Thus, the structure memory is not adequate for handling global structures. The main use of the memory system in our implementation scheme is for storing event lists. An event list is maintained by one process, the ECM, but various pointers to its elements may exist in the processor, in EHM's. There is no sense in copying parts of an event list as a consequence of a change in the list, such as deletion of an element, (as will happen if the structure memory is used) since an event list is global.

The reason for eliminating side effects from the structure memory is avoiding nondeterminacy in computations using it, as explained in [Ac-77]. In our implementation schemes the memory is used by managers and instances of event handlers in ways which do not cause nondeterminacy in addition to the nondeterminacy of the algorithms themselves;

the algorithms nondeterminacy is implied by the semantics of the language.

9.8.1 Storage Organization

The memory system handles free storage internally. The smallest storage allocation unit is a cell. A *cell* is a contiguous group of n memory words where n is some fixed (small) number. Whenever a storage area is needed, an operation packet is sent to one of the memory controllers requesting $k \geq 1$ cells. The memory controller handling the request creates a *block* of k cells by linking k cells it obtains from the free cell network in one of several fixed representations. These representations may include for example a linked list, or a tree of some fixed branching factor. A block in any such representation has a unique cell, called the *block head*, associated with it; the block head contains identifying information about the block and its use (e.g., as a list element). Once a block is formed, its words are addressed as if it is a contiguous storage area by specifying the block head address and an offset; the memory controller traces the cells constituting the block in order to access the desired word. The representation of memory blocks is selected by the compiler. For example, big event objects can be represented as trees whereas small ones as linked lists.

Handling the free storage can be done as described in [Ac-77]. Each memory controller has a free cell list associated with it (initially the whole memory space is divided among the controllers). Each memory controller always presents a free cell from its free cell list to the *free cell network* (unless its list is empty). Whenever it needs a cell it takes it from the free cell network. This network can be viewed as an arbitration network which passes free cell addresses from memory controllers to memory controllers. Whenever a memory controller decides to return a cell to the free storage it returns it to its private free cell list; thus, there is no need for any coordination among different memory controllers accessing free cell lists. In contrast to [Ac-77] where cells are returned to free storage on the basis of reference counts, in our case blocks are explicitly returned to free storage as described in chapter 7.

9.8.2 The Memory Controller

The main properties of the memory controller have been already described. We view it as a programmable processor which can be tailored exactly to the needs of our implementation schemes. The operation packets it understands include conventional instructions such as: write, read, read n words, or read-modify-write; instructions concerning free storage such as: allocate, or return; instructions supporting our representation of list elements described in chapter 7 such as increment a reference count; and instructions supporting requests that an ECM is asked to handle such as: insert, not-needed, book, cancel, acquire, or next, as defined in chapter 7.

Note that in order for a read-modify-write operation (e.g., test and set) to achieve the desired effect it should be executed as a nondivisible command by a memory module, and not just by a memory controller; otherwise, interleaving of command packets issued by several memory controllers may occur. Thus, we assume that a memory module can execute read-modify-write commands in addition to read or write commands.

The memory modules constitute a shared memory accessible to managers and to instances of event handlers. Thus, some of the operations on an event list which in our virtual system implementation scheme (described in chapter 7) are executed by ECM's can be executed directly by instances of event handlers; for example, read or not-needed. This decreases the number of packets per execution of an instance of an event handler.

When a memory controller receives a not-needed operation packet (indicating that a list element is no longer needed by the instance of the event handler) it can always return to free storage all the cells constituting the corresponding memory block except the block head (which contains information about the state of the list element and pointers to its neighbors). After doing so it can check whether the block head itself can be also returned to free storage according to the state of the element, using some nondivisible test and set instruction as described in chapter 7. Thus, storage associated with a list element can be partially freed even when it is still in the acquired state and before it enters the deleted state.

9.8.3 Congestion Deadlocks Due to the Memory System

The problem of congestion deadlocks arose in several places in this chapter. The solution employed was converting a schema to a safe schema by using signal tokens. The same problem may arise due to unsafe use of the memory system. The solution to this problem is identical to the solution of the first problem. Each access to the memory should be viewed as if done by a memory actor; the subschema containing this actor must be safe. From another point of view, room must be prepared for all possible results of a memory operation packet. The memory system does not spontaneously send packets destined to the instruction memory; thus, preparing room for the results is sufficient for preventing deadlocks due to congestions in the networks passing packets from the memory system to the instruction memory (the distribution network, and the control network in Figure 9.2).

An important question is: can deadlocks occur due to congestion of packets destined to the memory system in the arbitration network? In order to answer the question, first observe that an operation packet destined to the memory system does not specify a specific memory controller as its destination. Thus, such a packet is blocked only if all memory controllers are currently full (i.e., cannot accept new operation packets). Such a situation does not imply that a deadlock has occured since a memory controller can accept new operation packets after completing the processing of a previous one (it can process several packets concurrently). The only case in which a deadlock can arise is when no operation packet currently waiting in the memory controllers can be completely processed. This means that each operation packet either waits because a booked list element is encountered (in case of the packets next or book), or because no more free storage cells are available. In this situation the packets needed to allow completion of the operation packets currently waiting in memory controllers (cancel, acquire, or not-needed) cannot enter the memory system and a deadlock exists.

Our solution to the deadlock problem consists of the following algorithm executed by a memory controller when its input queue is full.

1. Move some of the currently waiting operation packets to a special buffer within the memory controller. If there is no room in the special buffer then goto step 2, else terminate.
2. Move some of the currently waiting operation packets to the memory itself after obtaining memory cells from the free cell network. If not enough free cells are available then goto step 3, else terminate.
3. Send some of the currently waiting operation packets back to the instruction

memory; these packets will arrive again later.

Step 3 causes the waiting packets to circulate in the system, thus allowing new operation packets to enter the memory system. It causes communication overhead and this is the reason why it is used as the last resort by a memory controller.

9.9 Performance Evaluation

Appendix C develops bounds on the performance of a certain class of EBL programs on the data flow processor. As in the case of a processor network the throughput of a program is used as a performance measure. The analysis assumes that the arbitration network and the distribution network consist of several routing networks. Objects propagate in the system subject to constraints on routing network capacity and operator capacity. As in the case of a processor network a linear programming problem whose solution yields the maximum possible throughput is defined. The analysis is done first without assuming a specific implementation scheme, and then taking into account our manager based implementation scheme.

9.10 Summary

Several schemes for implementation of EBL on a data flow processor have been developed in this chapter. Each of these schemes requires a memory system in addition to the instruction memory. The memory system handles free storage internally from efficiency reasons; free storage could be managed outside the memory system. Most of our schemes do not require additional mechanisms (except the memory system). Some of the schemes, the script copying schemes, require additional supporting mechanisms: the request network, and the mechanisms required by the underlying procedure implementation scheme.

Procedures recursion and semaphore operations can be easily expressed in EBL. Therefore, our implementation schemes actually add these capabilities to the processor. The data flow implementation schemes outlined in this chapter can be adapted to directly solve specific problems in the data flow processor. For example, in order to incorporate procedures in some data flow processor language, procedure managers can be created. A procedure manager can be viewed as a combination of an event handler manager and an event class manager. The event list maintained by the procedure manager can simply contain requests for procedure activations.

10. Conclusions and Directions for Further Research

This chapter summarizes the research, presents the conclusions, and suggests directions for further research.

10.1 Summary and Conclusions

The purpose of this research has been the development of a language for parallel programming in a distributed system environment, and the investigation of strategies for its implementation on multiple processor systems. Several alternatives for the underlying computational model have been considered. This dissertation has analyzed some of the similarities and the fundamental differences between our model, on one hand, and message passing models and process based models, on the other hand; and motivated the selection of event semantics as the underlying model.

The fundamental characteristic of our event model is the ability of an instance of a program unit (an instance of an event handler) to unilaterally broadcast messages (cause events) without specifying their targets. The receivers (event handlers) autonomously decide whether they are interested in the messages (the information about the nature of the occurrence of the events) or not. A receiver is capable of performing a powerful operation: it can remove from the ether, which encloses all program units, one or more messages in an atomic action. Each instance of an event handler only causes several events, each of which activating possibly several instances of event handlers (i.e., creating new activities) which cause new events; this is the way by which the computation proceeds in our model.

EBL is a nonprocedural language. As for every nonprocedural language, the implementation is required to select an algorithm to perform the operations indirectly specified by the program. Such an implementation may be inefficient especially if the language contains too powerful constructs. Therefore, the main objective of this research has been to devise a language which is general enough to allow expressing a wide variety of computations, but is restricted enough to enable efficient implementations. During the course of this research many constructs and features have been considered as candidates to be included in the language. In some cases constructs have been rejected since they are inherently difficult to implement or imply inefficient implementation (inefficient run time code). In other cases constructs have not been incorporated in the language simply due to our desire to concentrate in this research on the investigation of the fundamental characteristics of event semantics.

We have not tried to achieve a minimal language since we did not want to obtain a language which is difficult to use. Thus, the effects contributed by some of the language constructs can also be achieved via others. For example, tags are redundant since their effect can be achieved by a single global counter implemented as a **single_use recurrent** event class identifier. However, this alternative unnecessarily serializes the process of getting new tags and complicates programs.

Each construct or type in the language has an important role which justifies its existence. **single_use recurrent** events are the work horse of the language. They allow modeling of all conventional language constructs discussed in chapter 5. **single_use non_recurrent** events are useful in certain real time applications (e.g., for modeling an elevator push button), for representing sets which can grow and shrink, or for controlling mutual exclusion. multi_use recurrent events allow broadcasting a message to several receivers as well as modeling mutable database records (together with the last predicate). multi_use non_recurrent events allow computing functions by tables and modeling sets which cannot shrink. Tags provide for distinguishing between the effects of different instances of the same event handlers as well as joining several events belonging to the same logical computation. The predicates allow one to explicitly control the order in which instances of an event handler are activated. Modules have no dynamic effects; they contribute to the modularity of the language as discussed in chapter 3.

The contribution of our single_use events to the expressive power of the language is significant. In fact, they allowed us not to include in the language conventional constructs such as: variables, assignment statements, iteration constructs, procedures, functions, and semaphores. These constructs can be easily modeled in the language. In addition, events allow activation of parallel processes, synchronization of parallel processes, mutual exclusion, message passing, immutable objects, and the effect of mutable objects.

Even though mutable objects are not part of the language, their effect can be achieved in several unique ways. One way is to delete temporary objects (forget single_use events) belonging to several classes, and to create new objects (cause new events) from these classes. Another way is to create permanent objects (cause multi_use events) in certain classes, and to read the latest object from a class (using EBL's predicates).

The language design goals given in chapter 1 have been achieved: EBL is simple and its few constructs are quite primitive; nevertheless, it manifests many desired properties. Its expressive power is high as discussed in chapters 5 and 6. Modularity exists in the language in several forms as shown in chapter 3. Encapsulated program units and abstract data types can be created. Programs can be developed both in a top down design and in a bottom up manner. Parallelism in an EBL program is manifested in several levels as described in chapter 3. New activities can be easily spawned in a high rate, synchronized, and joined. These properties make EBL well suited as a language for parallel programming in a multiple processor system. The language does not contain constructs which are inherently difficult to implement or imply inefficient implementation.

The implementation schemes developed in this dissertation are uncommon in implementations of programming languages. The basic implementation scheme is especially designed for distributed systems; it involves many managers communicating with each other and operating without any centralized control. Unlike implementations involving one global manager the role of a manager in our scheme is limited; it can be either an event class manager or an event handler manager.

This thesis is yet another example of the fact that more difficulties are encountered in distributed system implementations than in the case of single processor system implementations; this is the price paid for achieving higher concurrency (as well as other advantages). An example of an operation that is more difficult to implement in a distributed system is locking of objects. We have developed a two phase locking (acquisition) algorithm in which deadlocks are prevented. In contrast to many existing algorithms which prevent deadlocks by defining a total order on all objects to be locked, our scheme only defines a partial order on all object classes. The advantage of this scheme is that objects can be locked by a requestor concurrently and not sequentially as in other algorithms.

The investigation of schemes for implementation of EBL on a processor network has led to several optimization problems involving distribution of objects in a network. These problems are of general interest and are not restricted to the context of EBL. We have proved that the optimization problems and even approximations to these optimizations are *NP*-hard, and suggested heuristic algorithms.

Various schemes for implementation of EBL on a data flow processor have been suggested. The existence of these schemes in addition to the network implementation schemes shows that the scope of the language is not restricted to a specific computer architecture, and that the language does not need special hardware to support its implementation. An implementation of EBL on a data flow processor actually adds to the processor certain capabilities which have been research subjects in the last several years (e.g., procedures and semaphores), since these can be easily expressed in the language.

One should note that implementations of the language (in particular our manager based implementation scheme) on geographically distributed systems are likely to be inefficient because of the delays which are inherent in such systems.

The language seems to be useful for applications involving, for example, application of resources (as can be seen from chapter 6). However, application of the language to specific domains can benefit from special constructs. For example, real time applications could make use of constructs which allow one to specify time constraints such as the maximum latency between events. However, schemes which guarantee these constraints must be devised.

10.2 Directions for Further Research

This research can be pursued both at the language level and at the implementation level. The language described in this thesis contains only a small set of constructs. These constructs allow one to model quite easily many other constructs but it is not realistic to assume that a programmer would like to explicitly translate such constructs each time they are used. In order to make the language more practical it should be supplemented by additional constructs. The approach of chapter 5 can be pursued, or the event semantics might be incorporated in some existing language.

A natural future step is an implementation of the language on some multiple processor system, e.g., the MuNet [Wa-78b]. Once the language is implemented its usability can be determined. Various measurements can then be performed to determine the overhead introduced by our managers and the efficiency of the manager based implementation scheme. Another way to evaluate the implementation scheme is by means of a simulation. Another research direction is to design a multiple processor architecture which directly supports the language.

There are issues which have not been addressed in the thesis and one may want to investigate how they affect the results of this research. Examples of such issues are: reliability of the underlying system, and concurrent execution of several programs on the same system.

Will multiple processors speak EBL? Many arguments favoring a positive answer have been given in this dissertation. However, the development of languages for parallel programming in a distributed system environment will not, and should not, terminate with this research. Additional languages, perhaps more attractive, can be expected.

11. References

[Ac-77] Ackerman, W. B., A Structure Memory for Data Flow Computers, *MIT/LCS TR-186*, Aug. 1977.

[Ah-77] Aho, A. V. and Ullman, J. D., Principles of Compiler Design, *Addison-Wesley Pub. Co., Reading Mass.*, 1977.

[Ba-75] Bayer, R., On the Integrity of Data Bases and Resource Locking, *Lecture Notes in Computer Science* 39, *Proc. 5th Inf. Symp.*, Sep. 1975.

[BH-74] Brinch Hansen, P., A Programming Methodology for Operating System Design, *Information Processing 74, North-Holland Pub. Co., Amsterdam*, 1974, 394-397.

[BH-75] Brinch Hansen, P., The Programming Language Concurrent Pascal, *IEEE Transactions on Software Engineering*, *Vol. SE-1*, *no.* 2, June 1975, 199-207.

[Ch-74a] Chamberlin, D. D. and Boyce, R. F., SEQUEL: A Structured English Query Language, *Proc. 1974 ACM SIGMOD Workshop on Data Description, Access and Control*, 1974.

[Ch-74b] Chamberlin, D. D. et al., A Deadlock-Free Scheme for Resource Locking in a Data-Base Environment, *Information Processing 74, North-Holland Pub. Co., Amsterdam*, 1974, 340-343.

[Co-71] Courtois, P. J. et al., Concurrent Control with Readers and Writers, *Comm. ACM 14, 10*, Oct. 1971, 667-668.

[De-73] Dennis, J. B. and Fossen, J. B., Introduction to Data Flow Schemas, *MIT/MAC Computation Structures Group Memo 81-1*, Sep. 1973.

[De-75] Dennis, J. B. and Misunas, D. P., A Preliminary Architecture for a Basic Data-Flow Processor, *2nd IEEE Symp. on Comp. Arch., N.Y.*, Jan. 1975, 126-132.

[De-77] Dennis, J. B. et al., A Highly Parallel Processor Using a Data Flow Machine Language, *MIT/LCS Computation Structures Group Memo 134*, Jan. 1977.

[Di-68a] Dijkstra, E. W., Co-operating Sequential Processes, in *Programming Languages, Ed. F. Genues, Academic Press, New York*, 1968.

[Di-68b] Dijkstra, E. W., The Structure of the "THE" Multiprogramming System, *Comm. ACM 11, 5*, May 1968, 341-346.

[Di-71] Dijkstra, E. W., Hierarchical Ordering of Sequential Processes, *Acta Informatica 1*, 1971, 115-138.

[Di-72] Dijkstra, E. W., Notes on Structured Programming, in *Structured Programming, Academic Press, New York*, 1972, 1-82.

[Ga-79] Garey, M. R. and Johnson, D. S., Computers and Intractability, A Guide to the Theory of NP-Completeness, *W. H. Freeman and Co., San Francisco*, 1979.

[Gr-75] Grief, I., Semantics of Communicating Parallel Processes, *MIT/MAC TR-154*, 1975.

[Gr-78] Gray, J., Notes on Data Base Operating Systems, *IBM Research Laboratory, San Jose Ca., RJ 2188*, Feb. 1978.

[Ha-78] Halstead, R. H., Multiple-Processor Implementations Of Message-Passing Systems, *MIT/LCS TR-198*, Jan. 1978.

[Ha-79] Halstead, R. H., Reference Tree Networks: Virtual Machine and Implementation, *MIT/LCS TR-222*, July 1979.

[He-69] Hewitt, C., PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot, *IJCAI-69, Washington, D. C.*, May 1969.

[He-73] Heart, F. E. et al., A New Minicomputer/Multiprocessor for the ARPA Network, *AFIPS Conference Proc., Vol. 42*, June 1973, 529-537.

[He-76] Hewitt, C., Viewing Control Structures as Patterns of Passing Messages, *MIT/AI*

Working Paper 92, Aug. 1976.

[He-77] Hewitt, C. and Baker, H., Laws for Communicating Parallel Processes, *MIT/AI Working Paper 134*, 1977.

[He-79] Hewitt, C. et al., Specifying and Proving Properties of Guardians for Distributed Systems, *Proceedings of the International Symposium on Semantics of Concurrent Computation, Evian France (in Lecture Notes in Computer Science 70, Springer-Verlag New York)*, July 1979.

[Ho-74] Hoare, C. A. R., Monitors: An Operating System Structuring Concept, *Comm. ACM 17, 10*, Oct. 1974, 549-557.

[Ho-78a] Hoare, C. A. R., Communicating Sequential Processes, *Comm. ACM 21, 8*, Aug. 1978, 666-677.

[Ho-78b] Horowitz, E. and Sahni, S., Fundamentals of Computer Algorithms, *Computer Science Press, Inc., Maryland*, 1978.

[Je-77] Jenny, C. J., Process Partitioning in Distributed Systems, *IBM Zurich Research laboratory RZ 873*, April 1977.

[Kn-75] Knuth, D. E., The Art of Computer Programming Vol. 1, *Addison-Wesley Pub. Co., Reading Mass.*, Feb. 1975.

[Ko-76] Koffler, R. P., An Event Based Automobile Oriented High Level Programming Language, *S.B. Thesis, Department of Electrical Engineering and Computer Science, MIT*, Aug. 1976.

[Ko-79] Kornfeld, W. A., Using Parallel Processing for Problem Solving, *S.M. Thesis, Department of Electrical Engineering and Computer Science, MIT*, May 1979. (A short version was published in *IJCAI-79*, Tokyo, Aug. 1979, 490-492.)

[La-78] Lamport, L., Time, Clocks, and the Ordering of Events in a Distributed System, *Comm. ACM 21*, 7, July 1978, 558-565.

[Le-75] Lesser, V. R. et al., Organization of the Hearsay II Speech Understanding System, *IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-23, no. 1*, Feb. 1975, 11-24.

[Le-79] Leung, C. K. C., ADL: An Architecture Description Language for Packet Communication Systems, *MIT/LCS Computation Structures Group Memo 185*, Oct. 1979.

[Mi-75] Misunas, D. P., Deadlock Avoidance in a Data-Flow Architecture, *MIT/MAC Computation Structures Group Memo 116*, Feb. 1975.

[Mi-77] Miranker, G. S., Implementation Schemes for Data Flow Procedures, *MIT/LCS Computation Structures Group Memo 138-1*, Feb. 1977.

[Mo-76] Modell, H. S. et al., Coordination of Parallel Processes in PL/1, *Proc. of the 1976 International Conference on Parallel Processing*, Aug. 1976, 247-253.

[Mo-77] Morgan, H. L. and Levin, K. D., Optimal Program and Data Locations in Computer Networks, *Comm. ACM 20*, 5, May 1977, 315-322.

[Na-63] Naur, P. et al., Revised Report on the Algorithmic Language ALGOL 60, *Comm. ACM* 6 (*1*), 1963, 1-17.

[Pa-71] Patil, S. S., Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination Among Processes, *MIT/MAC Computation Structures Group Memo 57*, Feb. 1971.

[Pe-77] Peterson, J. L., Petri Nets, *ACM Computing Surveys* 9, 3, Sep. 1977, 223-252.

[Pf-74] Pfister, G. F., The Computer Control of Changing Pictures, *MIT/MAC TR-135*, Sep. 1974.

[Pn-79] Pnueli, A., The Temporal Semantics of Concurrent Programs, *Proceedings of the*

International Symposium on Semantics of Concurrent Computation, Evian France (in Lecture Notes in Computer Science 70, Springer-Verlag New York), July 1979, 1-20.

[Re-77a] Reed, D. P. and Kanodia, R. K., Synchronization with Eventcounts and Sequencers, *MIT/LCS Computer System Research Division RFC 138*, Mar. 1977. (Also published in *Comm. ACM 22*, 2, Feb. 1979, 115-123.)

[Re-77b] Reingold, E. M. et al., Combinatorial Algorithms: Theory and Practice, *Prentice-Hall, Inc.*, 1977.

[Re-78] Reed, D. P., Naming and Synchronization in a Decentralized Computer System, *MIT/LCS TR-205*, Sep. 1978.

[St-74] Strachey., C. and Wadsworth, C. P., Continuations: A Mathematical Semantics for Handling Full Jumps, *Technical Monograph RPG-11, Oxford University Computing Laboratory*, Jan. 1974.

[St-78a] Stone, H. S. and Bokhari, S. H., Control of Distributed Processes, *Computer Vol. 11, No. 7*, July 1978, 97-106.

[St-78b] Stucki, M. J. et al., Coordinating Concurrent Access in a Distributed Database Architecture, *Fourth Workshop on Computer Architecture for Non-Numeric Processing, SIGARCH Vol. VII No. 2*, Aug. 1978, 60-64.

[Su-77] Sullivan, H. and Bashkow, T. R., A Large Scale, Homogeneous, Fully Distributed Parallel Machine, I and II, *Proc. of The 4th Annual Symposium on COMPUTER ARCHITECTURE*, March 1977, 105-117, 118-124.

[Wa-78a] Ward, S. A. and Halstead, R. H., A Syntactic Theory of Message Passing, *MIT/LCS, DSSR Internal memorandum*, April 1978.

[Wa-78b] Ward, S. A., The MuNet: A Multiprocessor Message-Passing System Architecture, *Proc. Seventh Texas Conf. on Computing Systems*, Oct. 1978.

[We-79] Weng, K. S., An Abstract Implementation for a Generalized Data Flow Language, *Ph.D Thesis, Department of Electrical Engineering and Computer Science, MIT*, May 1979.

[Wi-77a] Winston, P. H., Artificial Intelligence, *Addison-Wesley Pub. Co., Reading Mass.*, 1977.

[Wi-77b] Wirth, N., Modula: a Language for Modular Multiprogramming, *Software-Practice and Experience, Vol. 7*, 1977, 3-35.

[Wi-77c] Wirth, N., Towards a Discipline of Real-Time Programming, *Proc. of an ACM Conference on Language Design for Reliable Software, Raleigh, North Carolina*, March 1977, 190-195.

[Wo-77] Wong, K. C. and Edelberg, M., Interval Hierarchies and Their Application to Predicate Files, *ACM Tran. on Database Systems, Vol. 2, No. 3*, Sep. 1977, 223-232.

[Wu-72] Wulf, W. A. and Bell, C. G., C.mmp - A multi-mini-processor, *Proc. of the 1972 Fall Joint Computer Conference, Vol. 41*, 1972, 765-777.

Appendix A - The Formal Syntax

This appendix contains a formal definition of the syntax of EBL. The order of the sections is close to that of chapter 4 for a convenient reference. The notation is explained at the beginning of chapter 4.

A.1 Identifiers and Numbers

<ident> ::= <letter> { <letter> | <digit> | _ }

<unsigned_number> ::= { <digit> }$_1$

<character> ::= <digit> | <letter> | <special_character>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<letter> ::= a | ... | z | A | ... | Z

<special_character> ::= + | - | * | / | = | < | > | (|) | [|] | { | } | ...

A.2 Constants

<constant> ::= <unsigned_constant> | { + | - }$_1^1$ <unsigned_number>

<unsigned_constant> ::= <unsigned_number> | '<character> | **true** | **false**

A.3 Types and Type Identifiers

<type> ::= <simple_type> | <event_type> | <type_ident>

<simple_type> ::= <basic_type> | **tag**

<basic_type> ::= **int** | **bool** | **char**

```
<event_type> ::= { single_use | multi_use }1 { recurrent | non_recurrent }1
                          event { ( <type_list> ) }1

<type_list> ::= <type> { , <type> }

<type_ident> ::= <ident>
```

A.3.1 Type Identifier Definition

```
<type_definition> ::= <type_ident> { , <type_ident> } == <type_list> ;
```

A.4 Declarations

```
<declaration> ::=          <type_definition> |
                           <event_class_declaration> |
                           <tag_declaration> |
                           <event_handler> |
                           <module>
```

A.4.1 Event Class Identifier Declaration

```
<event_class_declaration> ::= <event_class_identifier>
                 { , <event_class_identifier> } : { <event_type> | <type_ident> }11 ;

<event_class_identifier> ::= <ident>
```

A.4.2 Tag Identifier Declaration

```
<tag_declaration> ::= <tag_ident> { , <tag_ident> } : tag ;

<tag_ident> ::= <ident>
```

A.5 Expressions

<exp> ::= <basic_exp> | <non_basic_exp>

<basic_exp> ::= <int_exp> | <bool_exp> | <char_exp>

A.5.1 Integer Expression

<int_exp> ::= { + | - }^{1} <int_term> { <int_adop> <int_term> }

<int_adop> ::= + | -

<int_term> ::= <int_factor> { <int_mop> <int_factor> }

<int_mop> ::= * | / | div | mod

<int_factor> ::= <unsigned_constant> | <formal_parameter> | (<int_exp>) |
abs (<int_exp>) | ord (<char_exp>)

A.5.2 Boolean Expression

<bool_exp> ::= <bool_term> { <bool_adop> <bool_term> }

<bool_adop> ::= or | xor

<bool_term> ::= <bool_factor> { <bool_mop> <bool_factor> }

<bool_mop> ::= and

<bool_factor> ::= { not }^{1} <bool_primary>

<bool_primary> ::= <constant> | <formal_parameter> | <bool_relation> |
(<bool_exp>)

<bool_relation> ::= <exp> { = | <> }$^1_1$ <exp> |
<int_exp> { < | <= | >= | > }$^1_1$ <int_exp>

A.5.3 Character Expression

<char_exp> ::= <constant> | <formal_parameter> | chr (<int_exp>)

A.5.4 Non-Basic Expression

<non_basic_exp> ::= <event_class_identifier> | <tag_ident> |
                                        <formal_parameter>

A.6 Event Handler

<event_handler> ::= on <event_handler_heading> <event_handler_body> end ;

A.6.1 Event Handler Heading

<event_handler_heading> ::= <event_descriptor_list> { where <condition> }^{1}

<event_descriptor_list> ::= <event_descriptor> { $\wedge$ <event_descriptor> }

<event_descriptor> ::= <class_class_identifier> { (<formal_parameter_list>) }^{1}

<formal_parameter_list> ::= <formal_paramctor_section>
                                        { , <formal_parameter_section> }

<formal_parameter_section> ::= <formal_parameter> { , <formal_parameter> }
                                                        : <type>

<formal_parameter> ::= <ident>

A.6.2 Where Clause Condition

```
<condition> ::= <bool_exp> |
                ( <bool_exp> ) { ∧ <predicate> $\}_1$ |
                <predicate> { ∧ <predicate> }

<predicate> ::= exist ( <event_class_identifier> ) |
                none ( <event_class_identifier> ) |
                min ( <int_exp> ) | max ( <int_exp> ) |
                first ( <event_descriptor> ) | last ( <event_descriptor> )
```

A.6.3 Event Handler Body

```
<event_handler_body> ::= { <tag_declaration> }
                         { par_cause | seq_cause $\}_1^1$ <script>

<script> ::= { <event> { ; <event> } $\}^1$

<event> ::= <class_designator>
                         { ( <actual_parameter> { , <actual_parameter> } ) $\}^1$

<class_designator> ::= <event_class_identifier> | <formal_parameter>

<actual_parameter> ::= <exp>
```

A.7 Module

```
<module> ::= module <module_interface> <module_body> end ;

<module_interface> ::=   { import: { <ident_list> | all $\}_1^1$ ; $\}^1$
                         { export: { <ident_list> | all $\}_1^1$ ; $\}^1$

<ident_list> ::= <ident> { , <ident> }

<module_body> ::= { <declaration> }
```

A.8 Program

```
<program> ::= <module_body>
```

Appendix B - Network Performance Evaluation

This appendix develops bounds on the performance of a certain class of EBL programs on a network. It first describes our view of the network and defines the class of programs captured by our model. Then, a method for computing performance bounds which is not based on a specific implementation scheme is presented. Finally, a method for computing performance bounds which takes into account our manager based implementation scheme is developed.

B.1 The Model

Our view of the network is basically as described in the beginning of chapter 8; however, we do not assume that all processors are identical. The links in this model are directed since we distinguish in the analysis between the flow of objects from node n to node n' and the flow of objects from node n' to node n. Each undirected link in the original network is therefore viewed as a pair of directed links in the model. The network imposes constraints on the rate in which computations can be performed. Our analysis takes into account two constraints: CPU capacity and link capacity. CPU capacity is derived from the speed in which the processor can execute instructions. The capacity of a link is derived from its bandwidth (its speed). Propogation delay along links is not taken into account; we assume it is negligible.

The network contains a set of input links L_{in} through which information from the external world arrives, and a set of output links L_{out} through which information is sent to the external world. The information entering the system through L_{in} consists of events from a distinguished event class denoted by e_{in}. The information leaving the system through L_{out} consists of events from another distinguished event class, denoted by e_{out}. For simplicity we assume that $e_{in} \neq e_{out}$.

We assume a special class of programs. Instances of event handlers are activated in response to occurrences of events from the class e_{in}. They perform computations by activating other instances of event handlers and cause events from the class e_{out}. All event class identifiers are of single_use recurrent types. The program uses events from the class e_{in} and causes (as output) events from the class e_{out}. As a performance measure of the program we will examine the *throughput* Z of the program: the rate in which events from the class e_{in} are used. Other performance measures are possible; e.g., the delay between the time an input event arrives and the time all related output events leave the system. Throughput, however, is easier to analyze and captures an important system aspect. Our throughput analysis can be applied to any EBL program satisfying the following conditions:

1. The program contains two distinct event class identifiers e_{in} and e_{out} as explained earlier.
2. All event class identifiers are of single_use recurrent types.
3. No formal parameters of type event appear as class designators in the script of any event handler.
4. For each event class identifier e, the total frequency in which events from class e are caused by the program or enter the system equals the total frequency in which events from class e are used by the program or leave the system.

Condition 1 can be easily relaxed; we shall not do so. Conditions 2 and 3 are relaxed later.

Condition 4 is needed; otherwise, either the program is not executed correctly, or events accumulate in the system without any bound on their number. The following program, for example, cannot be analyzed in our model:

```
on $e_{in}$
   par_cause  $e_{out}$ ; e
end ;
```

The reason is that over the long term the number of accumulated (unused) events from class e grows beyond any bound.

B.2 Maximum Possible Throughput

A method for computing a bound on the maximum possible throughput of a given program on a given network is developed now. (The techniques of this section are based on the analysis in [Ha-79].) Some qualifications on the method are given later. Several copies of the script of each event handler h exist in nodes of the network. Whenever an instance of an event handler is executed, event objects are created according to the script of the event handler. These event objects propagate through the network links until they are either used by an instance of some event handler or leave the system through the output links (if they belong to e_{out}).

Event objects propagate in the network subject to constraints on link capacity and CPU capacity. The constraints and the function to be maximized define a linear programming problem whose solution yields the maximum possible throughput. No higher throughput is possible since for each execution of an instance of an event handler only CPU time for preparing event objects is taken into account; no overhead of finding and acquiring matching event collections exists. Events move in the network in the optimal manner (yielding maximum throughput) as if they are guided by some all knowing power which consumes no computational resources. Table B.1 defines the terminology to be used in the sequel. Some of the identifiers are only used in later sections.

e	an event class identifier
e_{in} (e_{out})	the input (output) event class identifier
h	an event handler
e^h	e appears in the event descriptor list of h
L	the set of links in the network
l	a link in the network
L_{in} (L_{out})	the set of input (output) links: $L_{in} \subseteq L$ ($L_{out} \subseteq L$)
n, n'	nodes in the network
$I(n)$	the set of links entering node n
$O(n)$	the set of links leaving node n
C_{eh}	the number of events from class e caused by an instance of h
U_{eh}	the number of events from class e used by an instance of h
F_e	the frequency in which events from class e are caused
F_h	the frequency in which instances of h are executed
F_{xl}	the frequency (flow) of messages of class x over link l
F_{xn}	the frequency in which instances of x are executed on node n
P_{hn}	CPU time to execute an instance of h on node n
R_{xn}	CPU time to receive a message of class x on node n
S_{xn}	CPU time to send a message of class x from node n
T_{xl}	the time taken by a message of class x traversing link l
m_e	the ECM of e
m_h	the EHM of h
m, m_1, m_2	managers
M_{mn}	1 iff manager m resides on node n, else 0
Classes denoted by x:	
e, h, m_e, m_h	as defined earlier
$m_h n'$	a message from m_h destined to node n' for activating an instance of h
$m_e m_h$	a message from m_e to m_h (defined only if e^h)
$m_h m_e$	a message from m_h to m_e (defined only if e^h)
$m_e L_{out}$	a message from m_e to the output links

Table B.1 Network throughput terminology

The difference between the input flow of messages from class x to node n and the output flow of messages from class x from node n appears in several equations. The following function evaluates it:

$$D(F_{xl}) = \sum_{l \in I(n)} F_{xl} - \sum_{l \in O(n)} F_{xl}$$

The linear programming constraints are presented now. Conservation of events must hold for all e, n:

B.1 $$D(F_{el}) - \sum_{h} U_{eh}F_{hn} + \sum_{h} C_{eh}F_{hn} = 0$$

Link capacity cannot be exceeded; thus, for all l:

B.2 $$\sum_{e} T_{el}F_{el} \leq 1$$

The processing performed on node n (communication overhead and execution of instances of event handlers) cannot exceed the CPU capacity:

B.3 $$\sum_{e, l \in I(n)} R_{en}F_{el} + \sum_{e, l \in O(n)} S_{en}F_{el} + \sum_{h} P_{hn}F_{hn} \leq 1$$

Only events of class e_{in} can flow on input links:

B.4 $$F_{el} = 0 \quad \text{for all } e \neq e_{in} \text{ and } l \in L_{in}$$

Similarly, only events of class e_{out} can flow on output links:

B.5 $$F_{el} = 0 \quad \text{for all } e \neq e_{out} \text{ and } l \in L_{out}$$

All frequencies and flows must be non-negative; thus, for all e, h, l, n:

B.6 $$F_{el}, F_{hn} \geq 0$$

Finally, the objective function to be maximized is:

B.7 $$Z = \sum_{l \in L_{in}} F_{e_{in}l}$$

We do not claim that the above scheme yields the maximum possible throughput of a program over all possible implementation schemes of EBL on a network. One reason is that we have assumed that an instance of an event handler is activated and fully executed on the same node. Implementations in which the script itself is distributed in the network are possible. Our analysis can be extended to include such implementations as well. Another reason is that higher throughput for a given program may be achieved by optimizing the program (not its implementation); e.g., by eliminating redundant events, as discussed in chapter 7. Suppose, for example, the program consists of n event handlers of the form:

```
on $e_i$
   par_cause $e_{i+1}$
end ;
```

for $i=1, \ldots, n$, where $e_1=e_{in}$ and $e_{n+1}=e_{out}$. This program can be optimized (by a compiler) to the following one, thus yielding a higher throughput:

```
on $e_{in}$
   par_cause $e_{out}$
end ;
```

Thus, optimized programs may yield a throughput which is higher than the one suggested by our throughput analysis.

As a special case, the throughput analysis can be applied to the virtual system of chapter 7, assuming a fixed number of processors. Each link in the virtual system has an unlimited capacity; thus, we substitute in B.2 $T_{el}=0$ for all e, l. Also, there is no communication overhead in the virtual system; thus, we substitute in B.3 $R_{en}=S_{en}=0$ for all e, n.

It is interesting to examine how the linear programming approach handles data dependency. Where clauses are not reflected in the model. The optimal linear programming problem solution yields flows of events and frequencies of execution of instances of event handlers from which some data dependency properties can be deduced. The optimal solution assumes that the various events have the properties (parameters) needed for achieving the maximum throughput. Consider, for example, the following program for computing factorial:

```
on $e_{in}$ (n: int)                       { $h_1$ }
   par_cause  $e_1$ (n, n, 1)
end ;

on $e_1$ (n, i, p: int) where i≤1          { $h_2$ }
   par_cause  $e_{out}$ (n, p)             { p=factorial(n) }
end ;

on $e_1$ (n, i, p: int) where i>1          { $h_3$ }
   par_cause  $e_1$ (n, i-1, i*p)          { iterate }
end ;
```

In the optimal solution, the rates in which events are caused from each of the event classes e_{in}, e_1, e_{out} are all equal, say to F_m. The rates in which instances of h_1 or h_2 are activated are also equal to F_m. However, the rate in which instances of h_3 are activated is 0. The optimal solution, therefore, assumes that for each input event $e_{in}(n)$ either n=0 or n=1. The maximum throughput of the program really occurs in this case, but it may be not realistic to assume that all input events satisfy the above assumption.

Can we add information to the model; e.g., that n assumes values in the range 0-4 with equal probabilities? The answer is positive. According to the added information $F_{h_2} = \frac{1}{5}(1 + 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4}) F_{e_1} = \frac{37}{60} F_{e_1}$. Similarly, $F_{h_3} = (1 - \frac{37}{60}) F_{e_1} = \frac{23}{60} F_{e_1}$. The solution is to distinguish between events of class e_1 activating instances of h_2, and those activating instances of h_3.

The way to express such information in our model in a general case is as follows. Suppose e is an intermediate event class identifier (i.e., $e \neq e_{in}$ and $e \neq e_{out}$) which appears in the headings of $h_1, \ldots, h_n$, once in each event descriptor list, and it is known that

(1) $F_{h_i} = \alpha_i F_e$ for $i=1, \dots, n$ where $\alpha_i \geq 0$.

e can be replaced by a new set of event class identifiers $e_1, \dots, e_n$; e_i will replace e in the heading of h_i. If for the original program $C_{eh}=K$, it is now replaced by n constants $C_{e_ih}=K\alpha_i$. Each instance of h causes $K\alpha_i$ events from event class e_i instead of K events from event class e. The new linear programming solution is a more accurate estimate of the throughput, but it is no longer a bound on the original problem throughput. It is a bound on the throughput if (1) above is considered to be a given constraint.

K is an integer and $K\alpha_i$ is not necessarily an integer; this poses no difficulty in the analysis. Note that (2) the constants α_i satisfy $\sum_i \alpha_i = 1$, or $\sum_i F_{h_i} = F_e = 0$. The above follows from the following two observations: First, (3) $\sum_i F_{h_i} \leq F_e$ since each time an instance of h_i is activated it uses one event from event class e. Second, (4) $\sum_i F_{h_i} \geq F_e$; otherwise events from class e will accumulate in the system. Events cannot accumulate in our model; this can be seen by summing equation B.1 over all n. From (1), (3), and (4) we get $F_e = \sum_i F_{h_i} = F_e \sum_i \alpha_i$, or $F_e (\sum_i \alpha_i -1) = 0$; this explains claim (2) above.

Suppose h is an event handler with a script which specifies an event whose class designator f is a formal parameter of the event handler. f may be bound to different event class identifiers in different instances of h. How can the appropriate information about f be expressed in the linear programming constraints? Consider a simple case in which any instance of $h_1, \dots, h_n$ causes activation of an instance of h in which f is bound to e; and other event handlers activate instances of h in which f is bound to other event class identifiers. In such case, instances of h cause events from class e in a frequency (5) $\sum_i F_{h_i}$. This sum can be included in a conservation equation for events of class e.

In more general cases, one can find the set S of event class identifiers to which f can be bound. S can be found by a simple algorithm which uses transitive closure. Without such algorithm, an assumption that S contains all event class identifiers in the program can be made; this assumption can only result in a higher bound on the throughput. Suppose f can be bound to $e_1, \dots, e_m$; h can be replaced by m distinct event handlers $h_1, \dots, h_m$. Each h_i specifies in its script an event from class e_i instead of the event associated with the formal parameter f. The throughput of the original program cannot be higher than that of the modified program.

The above scheme can yield a throughput bound which is higher than the maximum possible throughput. A refinement of the above approach can be made if some control flow analysis (or another source of information such as the user) relates the frequencies in which f is bound to various event class identifiers to other frequency variables such as in (5) above. We shall not pursue this direction further.

B.3 Other Event Types

So far we have only dealt with single_use recurrent events. Let us examine the difficulties posed by other event types; starting with single_use non_recurrent events. If single_use non_recurrent events are treated as if they are single_use recurrent events, the solution to the linear programming problem may be smaller than the maximum possible throughput. The reason is that computational resources (CPU time and link capacity) may be allocated in the analysis to process events which in fact do not occur.

Suppose e is of a **single_use non_recurrent** type and an event of class e is specified in the script of h. Some way to describe the fact that an instance of h can cause zero or one events from class e is needed. The solution is quite simple. A new event handler h_1 is added; the difference between h and h_1 is that an instance of the latter does not cause an event from class e. The procedure can be repeated for all event handlers and all event class identifiers of **single_use non_recurrent** types. If the script of h specifies one event from each of n distinct **single_use non_recurrent** event classes, 2^n-1 new event handlers are added.

Another approach can be taken if some information is known about e. For example, C_{eh} (which originally was 1) can be replaced by $\alpha\leq1$ to reflect the fact that the rate in which instances of h cause events from class e is only αF_h.

Our model cannot handle **multi_use** events in general. The reason is that this model can only handle intermediate events which are used (consumed) by the program in the same rate in which they are caused. Since **multi_use** events are never consumed, the above condition can be only satisfied if both rates are zero. Suppose e is of a **multi_use** type. Simply selecting $U_{eh}=0$ for all h, and $C_{eh}\geq0$ (according to the script of h) will result in $F_h=0$ for each h satisfying $C_{eh}>0$.

In certain cases **multi_use** events can be handled in our model. Suppose all event handlers in the program are single_use event handlers and it is known that e can be implemented as a record variable (as described in chapter 7). If the script of h specifies $n>0$ events from class e and e appears in $m\geq0$ event descriptors in the event descriptor list of h, one can simply select $C_{eh}=n$ and $U_{eh}=n$ (independently of m). If, however, $n=0$ then one can select $C_{eh}=U_{eh}=0$.

B.4 The Effect of Managers

This section shows how maximum possible throughput of a given program on a given network can be calculated when our particular implementation scheme (including managers) is chosen. At first, the problem may seem not a simple one because of the interaction among the managers. However, simplifying assumptions can be made since we are interested in finding a bound on the throughput; these assumptions are made explicit in the sequel.

For each event class identifier e, there is a corresponding ECM m_e; similarly, for each event handler h, there is a corresponding EHM m_h. In this section we assume that each manager resides on one node of the network and never moves. The node on which each manager resides is obtained from the solution to the object distribution problem or from another source. Several copies of the script of an event handler h can exist in various nodes of the network; m_h can send an *activation message* to any of these nodes in order to activate an instance of h. The managers and the scripts interact in a well defined manner. Figure B.1 shows the possible interactions among managers and scripts for a program containing 2 event handlers and 3 event classes. Several scripts of each event handler exist.

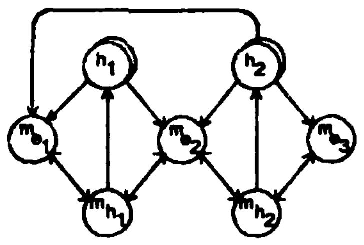


Figure B.1 The interactions among managers and scripts

An arc going from m_{h_i} to h_i represents activation messages. An arc going from h_i to m_{e_j} represents event objects from class e_j caused by instances of h_i. An arc connecting managers m_i and m_j represents messages exchanged between m_i and m_j. The maximum throughput is found similarly to section B.2. The difference is that messages to (from) managers and CPU requirements of managers are taken into account. The terminology for the following analysis is shown in Table B.1.

The total rate in which instances of h are activated is:

B.8 $$F_h = \sum_n F_{hn}$$

The total rate in which events from class e are caused is:

B.9 $$F_e = \sum_h C_{eh} F_h + \sum_{l \in L_{in}} F_{el}$$

Conservation equations can be written for the various types of messages in the model. Conservation of events must hold for all e, n:

B.10 $$D(F_{el}) + \sum_h C_{eh} F_{hn} - M_{m_e n} F_e = 0$$

Note the difference between equations B.1 and B.10. Here, the term $\sum_h U_{eh} F_{hn}$ does not appear since event objects from class e are sent to m_e and not directly to instances of event handlers. Activation messages from m_h to any node n' must be conserved; thus, for all h, n, n':

B.11 $$D(F_{m_h n' l}) + M_{m_h n} F_{hn'} - EQ(n,n')\, F_{hn'} = 0$$

where EQ(n,n')=1 iff n=n', else 0. Conservation of messages from an EHM to an ECM must hold; thus, for all e, h (e^h), n:

B.12 $$D(F_{m_h m_e l}) + M_{m_h n}(\alpha_1 F_h + \alpha_2 F_e) - M_{m_e n}(\alpha_1 F_h + \alpha_2 F_e) = 0$$

α_1 and α_2 are constants which can be selected according to the assumptions on the way m_h and m_e interact. α_1 represents the number of messages m_h sends to m_e in order to acquire an event it has previously selected. α_2 represents the number of messages m_h sends to m_e in order to check if an event of class e matches the heading of h. For the purpose of finding maximum throughput we can select α_2=0. A similar constraint on messages from an ECM to an EHM must hold; thus, for all e, h (e^h), n:

B.13 $$D(F_{m_e m_h l}) + M_{m_e n}(\beta_1 F_h + \beta_2 F_e) - M_{m_h n}(\beta_1 F_h + \beta_2 F_e) = 0$$

The roles of β_1 and β_2 are analogous to those of α_1 and α_2 respectively. In this model we *assume* that messages to the output links of the network are sent by $m_{e_{out}}$. Conservation of these messages must hold; thus, for all n:

B.14 $$D(F_{m_{e_{out}} L_{out} l}) + M_{m_{e_{out}} n} F_{m_{e_{out}} L_{out}} = 0$$

Since there is only one output event class, the following must hold:

B.15 $$F_{m_e L_{out}} = 0 \text{ for all } e \neq e_{out}$$

The rate in which events arrive to m_e must be equal to the sum of the rate in which they are used and the rate in which they are sent to the output links; thus, for all e:

B.16 $$F_e = \sum_h U_{eh} F_h + F_{m_e L_{out}}$$

The link capacity constraint now includes more terms than equation B.2; for all l:

B.17 $$\sum_e T_{el} F_{el} + \sum_{h,n'} T_{m_h n' l} F_{m_h n' l} + \sum_{h,e} T_{m_h m_e l} F_{m_h m_e l} + \sum_{e,h} T_{m_e m_h l} F_{m_e m_h l} + $$
$$+ T_{m_{e_{out}} L_{out} l} F_{m_{e_{out}} L_{out} l} \leq 1$$

Note that variables or constants of the form F_{xl}, T_{xl}, R_{xl}, S_{xl}, where x is $m_e m_h$ or $m_h m_e$ are only defined if $e \hat{} h$; thus, the qualification $e \hat{} h$ is not needed in the above inequality (and in similar cases in the sequel).

K(x) defines the communication overhead of receiving and sending messages of class x on node n.

$$K(x) = \sum_{x} \sum_{l \in I(n)} R_{xn} F_{xl} + \sum_{x} \sum_{l \in O(n)} S_{xn} F_{xl}$$

If x is a list (e.g., $m_h m_e$), $\sum_{x}$ above is a multiple summation. For each node n, the following CPU constraint should hold:

B.18 $$K(e) + K(m_h n') + K(m_h m_e) + K(m_e m_h) + K(m_{e_{out}} L_{out}) + \sum_{h} P_{hn} F_{hn} + \\ + \sum_{e} M_{m_e n} P_{m_e n} (F_e - F_{m_e L_{out}}) + \sum_{e} M_{m_e n} P_{m_e L_{out}} F_{m_e L_{out}} + \\ + \sum_{h} M_{m_h n} (P_{m_h hn} F_h + \sum_{e:\, e \hat{} h} P_{m_h en} F_e) \leq 1$$

The first 6 terms of B.18 should be clear now. The next two terms represent the load of all ECM's residing on node n. $P_{m_e n}$ is the CPU time to process an event from class e by m_e on node n including handling all relevant messages, if the event is used by the program. $P_{m_e L_{out}}$ is a similar coefficient associated with an event sent by m_e to the output links. The last term represents the load of all EHM's residing on node n. $P_{m_h hn}$ is the CPU time required by m_h on node n in order to acquire an event collection. $P_{m_h en}$ is the CPU time to process an event from class e by m_h on node n without acquiring it. For the purpose of finding the maximum throughput $P_{m_h en}$ can be selected as 0.

The flows on input links and output links are restricted as follows:

B.19 $$F_{xl} = 0 \quad \text{for all } x \neq e_{in} \text{ and } l \in L_{in}$$

B.20 $$F_{xl} = 0 \quad \text{for all } x \neq m_{e_{out}} L_{out} \text{ and } l \in L_{out}$$

All frequencies and flows must be non-negative; thus, for all e, h, l, n, n':

B.21 $$F_{hn}, F_{el}, F_{m_h n' l}, F_{m_h m_e l}, F_{m_e m_h l}, F_{m_{e_{out}} L_{out} l} \geq 0$$

The objective function to be maximized is:

B.22 $$Z = \sum_{l \in L_{in}} F_{e_{in} l}$$

In order to obtain a throughput bound which is closer to the actual throughput, more constraints can be added to the above set of constraints. Such constraints can describe limitations of a specific implementation of a manager (e.g., the maximum frequency in which it can iterate), or available memory on a node. The various extensions to the basic throughput analysis discussed in the previous sections can also be applied to the analysis of this section.

B.5 Throughput Analysis Summary

It is interesting to compare the maximum possible throughput of a program obtained without our specific implementation (section B.2) with that obtained if the effect of managers is taken into account (section B.4). This could show how much of the possible throughput is lost due to our managers overhead. Unfortunately, in both cases the maximum throughput cannot be expressed as a closed formula in general; thus, such a comparison can be made only by solving the linear programming problems and comparing the obtained values.

The linear programming approach can be used not only for analyzing a given program on a given network, but also for analyzing the effects of varying some of the system parameters on the maximum throughput of a program. For example, the effect of adding (removing) a processor or a link to the network, changing the capacity of a link, changing the speed of a processor, totally changing the network topology, or modifying the algorithm of an EHM or an ECM can be found.

Appendix C - Data Flow Performance Evaluation

This appendix develops performance bounds for a limited class of EBL programs on the data flow processor. The class of programs captured by the following discussion is identical to that of section B.1. First, a method for computing bounds on the maximum possible throughput which is not based on a specific implementation scheme is presented. Then, a method which specifically refers to our manager based implementation scheme is given. The methods are similar to those given in appendix B for a processor network.

For the analysis, the processor will be represented by the model of Figure C.1.

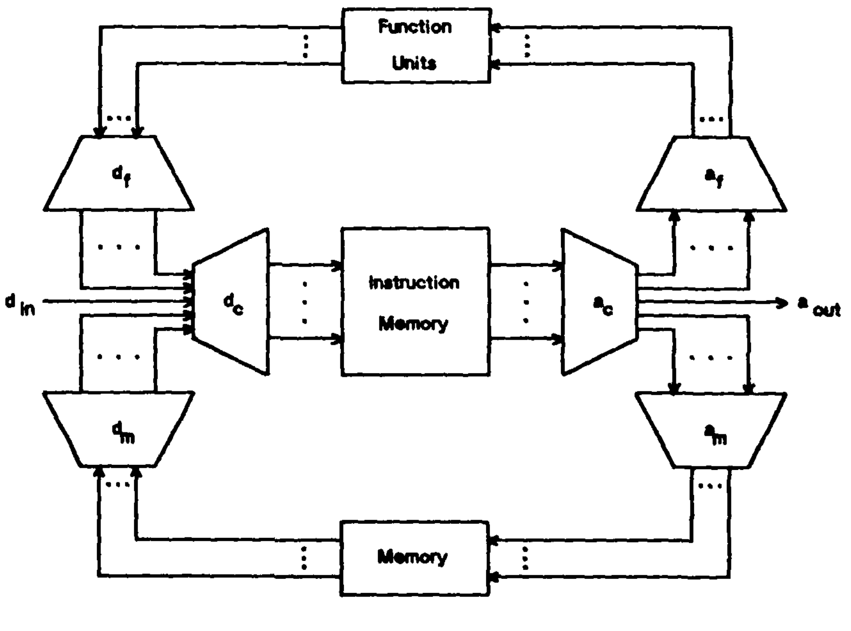


Figure C.1 The routing networks

We assume that the arbitration network consists of three parts: a_c, a_f, and a_m; and similarly, the distribution network consists of three parts: d_c, d_f, and d_m. We will call the above sub-networks *routing networks*. Input events enter the distribution network at d_{in};

output events leave the arbitration network at a_{out}.

Most of the terminology is similar to that used in appendix B. Additional terminology is defined in Table C.1.

f	an operator performing a fixed function (e.g., addition, memory access)
n_f	the number of identical units of f in the system
r	a routing network ($r=a_c$, a_f, a_m, d_c, d_f, d_m)
$F_{ed_{in}}$, $F_{m_ed_{in}}$	the frequency of events of class e entering at d_{in} (destined to m_e)
$F_{ea_{out}}$, $F_{m_ea_{out}}$	the frequency of events of class e leaving at a_{out} (sent by m_e)
T_{xr}	the total time taken by all packets related to: (1) a message of class x (x=e, m_ed_{in}, m_ea_{out}, or m_ee), or (2) all messages required for the execution of an instance of x (x=h, m_he, or m_hh), traversing routing network r
P_{xf}	the total time required for processing all packets related to: (1) a message of class x (x=e, m_ed_{in}, m_ea_{out}, or m_ee), or (2) the execution of an instance of x (x=h, m_he, or m_hh), on operator f
Classes denoted by x:	
e, h	as defined earlier
m_ed_{in}	an event of class e arriving from d_{in} destined to m_e
m_ea_{out}	an event of class e sent by m_e to a_{out}
m_ee	an event of class e caused by an instance of some event handler (i.e., not arriving from d_{in}) and used by an instance of some event handler (i.e., not sent to a_{out})
m_he	processing an event from class e by m_h ($e\hat{}h$) without acquiring it
m_hh	acquiring an event collection by m_h

Table C.1 Data flow throughput terminology

C.1 Maximum Possible Throughput

This section starts by showing how bounds on the maximum possible throughput can be computed. It first assumes that enough copies of the script of each event handler exist in the instruction memory. This assumption is *relaxed later in this section.* The constraints are analogous to those of appendix B. The frequency in which events of class e are caused equals the sum of the frequency in which events of class e are used and the frequency in which events of class e leave the system; thus, for all e:

C.1 $$F_e = \sum_h C_{eh} F_h + F_{ed_{in}}$$

C.2 $$F_e = \sum_h U_{eh} F_h + F_{ea_{out}}$$

The constraints on the routing networks are:

C.3 $$\sum_e T_{er}(F_e - F_{ea_{out}}) \leq 1 \quad \text{for } r = a_m, d_m$$

C.4 $$\sum_h T_{hr} F_h \leq 1 \quad \text{for } r = a_f, d_f$$

C.5 $$\sum_e T_{ea_c} F_e + \sum_h T_{ha_c} F_h \leq 1$$

C.6 $$\sum_e T_{ed_c}(F_e - F_{ea_{out}}) + \sum_h T_{hd_c} F_h \leq 1$$

The total capacity of the n_f operators of type f must not be exceeded; thus, for all f:

C.7 $$\sum_h P_{hf} F_h \leq n_f$$

The memory is viewed as an operator. Events arriving through d_{in} must be of class e_{in}:

C.8 $$F_{ed_{in}} = 0 \quad \text{for all } e \neq e_{in}$$

Similarly, events leaving through a_{out} must be of class e_{out}:

C.9 $$F_{ea_{out}} = 0 \quad \text{for all } e \neq e_{out}$$

All frequencies and flows must be non-negative; thus, for all e, h:

C.10 $$F_h, F_e, F_{ed_{in}}, F_{ea_{out}} \geq 0$$

The objective function to be maximized is:

C.11 $$Z = F_{e_{in} d_{in}}$$

The above analysis assumes no limitation on the number of copies of the script of an event handler. If there are n_h copies of the script of h, executed as n_h pipelines, then the following constraint must be added for all h:

C.12 $$F_h \leq n_h F_{h\ max}$$

$F_{h\ max}$ is the maximum frequency in which one copy of the script of h can be executed in pipeline mode. $F_{h\ max}$ can be computed for any given h if minimum delay times of the routing networks and minimum execution times of all operators are known.

C.2 The Effect of Managers

Maximum possible throughput when managers are taken into account can be computed analogously to the scheme of section B.4. The terminology is defined in Table C.1. Like equations C.1 and C.2, the frequency in which events of class e are caused equals the total frequency in which they are used or leave the system; thus, for all e:

C.13 $$F_e = \sum_h C_{eh} F_h + F_{m_e d_{in}}$$

C.14 $$F_e = \sum_h U_{eh} F_h + F_{m_e a_{out}}$$

There are constraints on the six routing networks; thus, for r = a_C, a_f, a_m, d_C, d_f, d_m:

C.15 $$\sum_e T_{m_e er}(F_e - F_{m_e d_{in}} - F_{m_e a_{out}}) + \sum_e T_{m_e d_{in} r} F_{m_e d_{in}} + \sum_e T_{m_e a_{out} r} F_{m_e a_{out}} +$$
$$+ \sum_h (T_{m_h hr} F_h + \sum_{e:\ e\hat{}h} T_{m_h er} F_e) + \sum_h T_{hr} F_h \leq 1$$

The first three terms represent activities of all ECM's; the fourth term represents activities of all EHM's; and the last term represents activities of all instances of event handlers. The constraint on the n_f operators of type f contains similar terms; for all f:

C.16 $$\sum_{e} P_{m_e ef}(F_e - F_{m_e d_{in}} - F_{m_e a_{out}}) + \sum_{e} P_{m_e d_{in} f} F_{m_e d_{in}} + \sum_{e} P_{m_e a_{out} f} F_{m_e a_{out}} +$$

$$+ \sum_{h} (P_{m_h hf} F_h + \sum_{e:\, e\hat{}h} P_{m_h ef} F_e) + \sum_{h} P_{hf} F_h \leq n_f$$

The constraints on input and output from the system are:

C.17 $$F_{m_e d_{in}} = 0 \quad \text{for all } e \neq e_{in}$$

C.18 $$F_{m_e a_{out}} = 0 \quad \text{for all } e \neq e_{out}$$

All frequencies and flows must be non-negative; thus, for all e, h:

C.19 $$F_h, F_e, F_{m_e d_{in}}, F_{m_e a_{out}} \geq 0$$

The objective function to be maximized is:

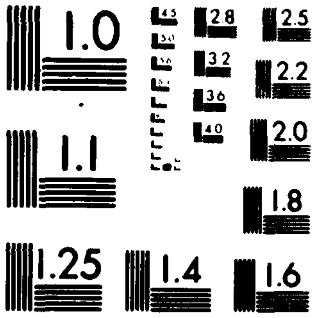
C.20 $$Z = F_{m_{e_{in}} d_{in}}$$

As in the previous section, more constraints can be added to describe further limitations imposed by a specific implementation scheme. For the *n* pipelines scheme, (like C.12) for all h:

C.21 $$F_h \leq n_h F_{h\,max}$$

C.3 Throughput Analysis Summary

The linear programming approach can be used not only for analyzing a given program on a given data flow processor, but also for analyzing the effects of varying some of the system parameters on the maximum possible throughput of a program. For example, the effect of adding (removing) an operator to the processor, changing the capacity of a routing network, or modifying the algorithm of an EHM or an ECM can be found. The various extensions to the basic throughput analysis discussed in appendix B can also be applied to the throughput analysis of this appendix. It is interesting to compare the maximum possible throughput of a program on a network with the maximum possible throughput of the program


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0

on the data flow processor. Unfortunately, such a comparison can be made only by solving the linear programming problems and comparing the obtained values.

Biographical Note

Asher Reuveni was born in Kefar-Saba, Israel on November 18, 1947. He grew up in Natanya Israel, where he graduated from Tcharnihovsky High School in 1965. From 1965 to 1971 he has attended the Technion, Israel Institute of Technology. In August 1969 he received the B.S. degree from the Department of Electrical Engineering in the Technion, and in July 1971 he received the M.S. degree from the Department of Electrical Engineering in the Technion. During 1969 to 1971 he was a teaching assistant in the Technion. He has served in Israel Defense Forces from August 1971 to July 1976.

Mr. Reuveni arrived in the U.S., with his wife Sara and their sons Rony (now 7) and Guy (now 3), in August 1976. During his doctoral studies at M.I.T. he was supported as a research assistant in the Real Time Systems Group of the M.I.T. Laboratory for Computer Science. During this period he has also worked part time at the B.B.N. (Bolt, Beranek and Newman) company. He received the Ph.D. Degree from the Department of Electrical Engineering and Computer Science in the M.I.T. in February 1980.

OFFICIAL DISTRIBUTION LIST

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314
12 copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217
2 copies

Office of Naval Research
Branch Office/Boston
Building 114, Section D
666 Summer Street
Boston, MA 02210
1 copy

Office of Naval Research
Branch Office/Chicago
536 South Clark Street
Chicago, IL 60605
1 copy

Office of Naval Research
Branch Office/Pasadena
1030 East Green Street
Pasadena, CA 91106
1 copy

New York Area
715 Broadway - 5th floor
New York, N. Y. 10003
1 copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D. C. 20375
6 copies

Assistant Chief for Technology
Office of Naval Research
Code 200
Arlington, VA 22217
1 copy

Office of Naval Research
Code 455
Arlington, VA 22217
1 copy

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
(Code RD-1)
Washington, D. C. 20380
1 copy

Office of Naval Research
Code 458
Arlington, VA 22217
1 copy

Naval Ocean Systems Center,Code 91
Headquarters-Computer Sciences &
Simulation Department
San Diego, CA 92152
Mr. Lloyd Z. Maudlin
1 copy

Mr. E. H. Gleissner
Naval Ship Research & Development Center
Computation & Math Department
Bethesda, MD 20084
1 copy

Captain Grace M. Hopper (008)
Naval Data Automation Command
Washington Navy Yard
Building 166
Washington, D. C. 20374
1 copy

Mr. Kin B. Thompson
Technical Director
Information Systems Division
(OP-91T)
Office of Chief of Naval Operations
Washington, D. C. 20350
1 copy